Salesforce APEX

Implemente soluções com padrões e técnicas de Orientação a Objetos

Casa do Código | alura

EDUARDO CARVALHO

Sumário

- ISBN
- Agradecimentos
- A quem se destina este livro?
- Quem sou eu?
- O que é a plataforma Salesforce?
- Parte 1: Introdução à linguagem APEX
  - 1 Preparando sua estação de trabalho
  - 2 Introdução a Orientação a Objetos
  - 3 Acessando dados através do APEX
  - 4 Capture as atualizações do banco de dados
  - 5 Teste tudo!
- Parte 2: Bem-vindo(a) ao lado Orientado a Objetos da Força
  - 6 Definindo papéis e responsabilidades
  - 7 Vamos adicionar novos Jutsus com a Orientação a Objetos
  - 8 Aplicando os novos Jutsus
- Parte 3: Trabalhando com serviços remotos
  - 9 Consumindo serviços REST
  - 10 Expondo serviços REST
- Parte 4: Trabalhando de forma assíncrona
  - 11 Alternativas de processamento assíncrono
  - 12 Trabalhe com classes agendadas
  - 13 Trabalhando com alto volume de dados
- Parte 5: Seja um(a) ninja nos testes
  - 14 F.I.R.S.T.
  - 15 Mocks
  - 16 Utilizando o framework Mocker
  - 17 Mockando tudo
  - 18 Conclusão
  - 19 Referências

ISBN

Impresso: 978-85-5519-363-7

Digital: 978-85-5519-362-0

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Carvalho, Eduardo
Salesforce APEX : implemente soluções com padrões e técnicas de orientação a objetos / Eduardo Carvalho. -- São Paulo : Aovs Sistemas de Informática, 2024.

ISBN 978-85-5519-363-7

1. Ciência da computação 2. Linguagem de programação 3. Programação (Computadores) 4. Software - Desenvolvimento - Testes I. Título.

24-189968 CDD-004

Índices para catálogo sistemático:

1. Ciência da computação 004

Tábata Alves da Silva - Bibliotecária - CRB-8/9253

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

Agradecimentos

Este livro foi um grande desafio no que se refere a estrutura, organização de conteúdo e gestão do tempo. Eu não teria conseguido finalizá-lo sem o apoio da minha esposa, Renata, que tanto amo e que, por muitas vezes, foi minha primeira revisora, e dos meus filhos, Eduardo e Rafael, que também sofreram com o "leia este trecho e veja se está bom". :)

Agradeço a meu pai, minha referência, pelo apoio e revisão. Não poderia esquecer de agradecer às minhas irmãs pelo apoio.

Agradeço aos meus amigos e companheiros da TOPi/BRQ, onde tive oportunidade e liberdade de aplicar e evoluir todas as técnicas aplicadas nesse livro.

Gostaria de destacar um agradecimento especial ao amigo Thiago Feitosa, que me estimulou de forma indireta durante a escrita deste livro perguntando-me sobre literatura voltada ao **Desenvolvimento Salesforce**, que foi o grande gatilho que fez com que eu iniciasse a minha jornada de escrita.

A quem se destina este livro?

Este livro é destinado a estudantes de tecnologia, desenvolvedores/desenvolvedoras que desejam entrar para o universo Salesforce e, também, a quem já está no mercado de trabalho e busca mais conhecimento sobre práticas de desenvolvimento e padrões na **plataforma Salesforce**. Através deste livro, leitores e leitoras serão capazes de criar soluções elegantes e escaláveis, destacando-se no mercado.

Quais conhecimentos você vai adquirir após ler este livro?

Este livro vai além de ensinar a desenvolver na plataforma Salesforce através da linguagem APEX. Ele proverá a você conhecimentos sobre:

- Orientação a Objetos;
- Práticas de Código Limpo (*Clean Code*);
- Princípios de desenvolvimento de software S.O.L.I.D.;
- Princípios de testes;
- Melhores práticas de desenvolvimento orientado a testes;
- Diversos padrões de projetos e boas práticas de desenvolvimento voltados para a Salesforce.

Note o volume de conteúdo voltado para padrões e princípios de desenvolvimento de software. Eles são a base para sua formação como **desenvolvedor(a)** e são independentes de linguagem.

Como devo estudar?

Este livro é um complemento do material fornecido pelo **Trailhead** (https://trailhead.salesforce.com/), no qual recomendo que crie sua conta e faça as trilhas de desenvolvimento, para que você possa conhecer mais sobre a plataforma Salesforce e seus recursos, começando por esta: https://trailhead.salesforce.com/users/strailhead/trailmixes/build-your-developer-career-on-salesforce.

Este livro guiará você de forma gradual: os exemplos iniciais são mais simples, justamente para que você possa se adaptar à linguagem e ao ambiente de desenvolvimento, passando da sintaxe básica da linguagem a conceitos de Orientação a Objetos, do básico ao avançado, para que você possa entender e realizar a evolução natural do código.

Gosto de dizer que os códigos iniciais não são produtivos, ou seja, você não deve utilizá-los em seus projetos, pois serão menos elegantes; porém, de forma gradual, aplicaremos padrões de projetos e práticas de desenvolvimento para que, ao final do livro, você tenha um código "bonito, cheiroso e produtivo".

Quem sou eu?

Meu nome é Eduardo Ribeiro de Carvalho, conhecido como "Bisso". Atuo na Zup Innovation como "Salesforce Solution Engineer" — um nome bonito para *Arquiteto* —, sendo o principal foco do meu trabalho formar o time de desenvolvimento e fornecer estruturas, padrões e frameworks para otimizar o trabalho do time.

Comecei minha carreira por volta de 1997, e desde então atuei com diversas tecnologias como VB, Java, WebMethods, SAP, Oracle e Salesforce, sendo que a minha principal formação foi em Java, com a qual atuo desde 2000. Em 2006 iniciei no que acredito que ser a minha maior paixão: lecionar. Formei dezenas de turmas em Java pela Global Code.

Desde 2010 venho atuando com a plataforma Salesforce, como desenvolvedor e líder técnico, onde pude trazer a bagagem do desenvolvimento Java, aplicando *Clean Code*, princípios do S.O.L.I.D., padrões de projeto e *Domain-Driven Design* para plataforma.

Por fim, gosto de dizer que sou um eterno aprendiz, e estou sempre buscando formas de deixar o código "bonito e cheiroso".

O que é a plataforma Salesforce?

A plataforma Salesforce é a maior referência em CRM, considerada pelo Gartner a líder no setor por 13 anos consecutivos (2008 a 2021). Este mercado está em expansão e carente de bons profissionais que saibam como desenvolver na plataforma. De acordo com a Salesforce, existe uma demanda de aproximadamente 4,2 milhões de novos empregos voltados para o ecossistema.

Por que aprender Salesforce?

Se olharmos somente pela ótica da empregabilidade, a Salesforce já seria uma ótima escolha, porém não é somente isso. A plataforma Salesforce envolve dezenas de soluções, desde a implementação na própria plataforma até a implementação de plugins para o *Slack*. Esse ecossistema é o que chamamos de visão 360º do cliente (*Customer 360*), em que todas as soluções são integradas para prover uma visão completa de todo o negócio do cliente.

Aqui está uma visão geral de todas as soluções da **plataforma Salesforce**:

- ***Sales Cloud***: Onde as empresas podem acompanhar a prospecção de novos clientes, monitorar resultados e automatizar processos de vendas;
- ***Service Cloud***: Fornece uma visão completa de todo o pós-venda do cliente, com capacidade de estruturar e automatizar *call centers*, fornecendo multicanais de atendimento e monitoramento do nível de serviço ao cliente;
- ***Marketing Cloud***: Inspira ação e fidelidade, humanizando cada momento para cada cliente. Potencializa todas as

interações com IA, de anúncios a marketing baseado em contas, de e-mail a comércio eletrônico, de redes sociais a serviços;
- ***Commerce Cloud***: Fornece experiência de comércio eletrônico, permitindo a entrada no mercado com mais rapidez e inteligência e proporcionando experiências personalizadas ao cliente em plataformas móveis, digitais e sociais;

- ***Integration***: O *MuleSoft* traz a capacidade de integrar sistemas do cliente com Salesforce de forma simples e rápida;

- ***Analytics***: O *Tableau* aprimora o Customer 360 com aplicativos de análise, modelos de painel e conexões de produtos perfeitas. Equipes de toda a empresa podem aprender mais facilmente com seus dados diretamente no fluxo de trabalho.

Através dessa visão do ecossistema, observe que existem dezenas de possibilidades de crescimento profissional e, também, de **empreender**, dado o volume de empresas que implementam soluções inteiras para a plataforma e as disponibilizam na loja de aplicativos App Exchange (https://appexchange.salesforce.com/).

Mas o que é um CRM? E o que faz?

CRM é a sigla usada para *Customer Relationship Management* (Gestão de Relacionamento com o Cliente). Porém, a definição de CRM vai além: é uma gestão 360º de vendas, marketing, atendimento e de todos os pontos de contato.

Estamos na era do cliente, da transformação digital, das novas tecnologias. Nesse cenário, o relacionamento também evoluiu, levando a um novo conceito também conhecido como *experiência do cliente*. O conceito de CRM significa ser centrado no cliente. É estratégia, é um processo, é ferramenta e tecnologia.

O CRM armazena informações de clientes atuais e potenciais, como nome, endereço, número de telefone, suas atividades e pontos de contato com a empresa, incluindo visitas a sites, ligações telefônicas, e-mails, entre outras interações. Entretanto, o software de CRM não é apenas uma lista de contatos elaborada: ele reúne e integra dados valiosos para preparar e atualizar suas equipes com informações pessoais dos clientes, histórico e preferência de compras.

Agora que você já viu a pontinha do iceberg, vamos nos aprofundar nesse novo universo que é a plataforma Salesforce, desenvolvendo de forma "elegante, bonita e cheirosa" :).

Parte 1: Introdução à linguagem APEX

A plataforma Salesforce criou a linguagem APEX com a intenção de aumentar a capacidade e escalar suas funcionalidades. Através dela, é possível criar aplicações, produtos e soluções completas, tudo isso utilizando a infraestrutura e o poder provido pela plataforma. O APEX é uma linguagem orientada a objetos que nasceu utilizando o Java como modelo, com algumas características do C#, sendo que a proporção seria de 95% Java e o restante, de C#.

Você verá muita similaridade com o Java. Dada essa característica e proximidade, vamos utilizar as convenções dessa linguagem para melhorarmos a nossa comunicação. Essas convenções serão apresentadas conforme a necessidade.

Nesta parte do livro, veremos exemplos mais simples, em português, para que você possa entender como funciona a linguagem APEX e como utilizar os seus recursos. Nesta parte também teremos um pouco de teoria, com a intenção de balizar conceitos de Orientação a Objetos, sem nos preocuparmos ainda em aplicar as melhores técnicas e práticas de programação.

A intenção nesse momento é que você compreenda a linguagem APEX, não a melhor forma de usá-la. Veremos isso mais adiante, com exemplos de códigos de fato produtivos que você poderá utilizar em sua aplicação, abordando práticas de *Clean Code*, TDD e padrões de projeto. Segure a ansiedade e siga com calma, que logo você terá muito, mas muito *code*!

Capítulo 1
Preparando sua estação de trabalho

A melhor maneira de você aumentar a sua capacidade de absorção do conteúdo deste livro é implementando todos os códigos

existentes nele, então ter um ambiente de desenvolvimento configurado é essencial. Para isso, montei um guia rápido com as principais ferramentas que você precisa para desenvolver profissionalmente em Salesforce.

Instalando o Git

A primeira ferramenta que você precisa ter para desenvolver profissionalmente é o gerenciador de versões de código **Git**. Ele é a ferramenta mais utilizada na atualidade e é essencial que você a domine. Todos os códigos do livro estão em um repositório Git, para que você execute os exemplos. Para isso, faça o download no site https://git-scm.com/downloads, conforme o seu sistema operacional, e siga o fluxo padrão de instalação.

Instalando o Java Developer Kit

Outra ferramenta necessária é o **Java Developer Kit** (JDK). Aqui vamos utilizar a versão 17 do JDK. Faça o download no site da Oracle (https://www.oracle.com/java/technologies/downloads/) e instale a versão 17, conforme o seu sistema operacional. Após a instalação, você precisará configurar a variável de ambiente `JAVA_HOME` e adicionar o Java no Path, de acordo com o seu sistema operacional.

Configurando o `JAVA_HOME` no Windows

A variável `JAVA_HOME` corresponde ao caminho onde o JDK foi instalado. Geralmente, fica em `C:\Program Files\Java\jdk-17.0.3.1`, mas esse nome pode variar de acordo com a versão instalada. Para configurar, você deve seguir alguns passos:

1. Acesse o *Meu computador*, clique com o botão direito do mouse e escolha *Propriedades*;
2. Na janela *Propriedades*, selecione a aba *Avançado* e clique em *Variáveis de ambiente*;

3. Na janela de *Variáveis de Ambiente*, clique no botão "*Novo...*" e adicione a variável `JAVA_HOME` com o caminho da instalação do JDK, recém-instalado;
4. Configure o *Path* do Windows, adicionando a instrução `%JAVA_HOME%\bin` no final da variável.

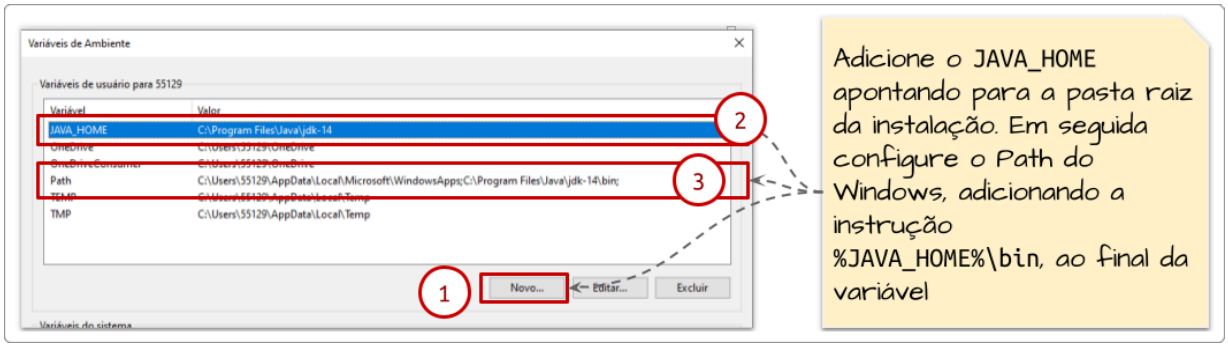


Figura 1.1: Configurando o Java Home

Para avaliar a sua instalação, execute no seu terminal o comando `echo %JAVA_HOME%`. Isso deverá imprimir o caminho do JDK instalado, o qual foi configurado em suas variáveis de ambiente. Em seguida, execute o comando `javac -version` no terminal para verificar se o JDK foi instalado corretamente.

Instalando o Node.js

Para instalar o **Node.js**, acesse o site https://nodejs.org/en/ e instale a versão LTS, seguindo as instruções de instalação conforme o seu sistema operacional. Para avaliar a sua instalação, execute o comando `node --version` no seu terminal, o que deverá imprimir a versão do Node.js instalada.

Instalando a Salesforce CLI

Após a instalação do Node.js, instale a ferramenta de linha de comando **Salesforce CLI**, disponível no site https://developer.salesforce.com/tools/sfdxcli. Realize a instalação conforme o seu sistema operacional. Para avaliar a sua instalação,

execute o comando `sfdx --version` no seu terminal, o que deverá imprimir a versão do Salesforce CLI instalada.

Instalando o VSCode

Para instalar o **VSCode**, acesse o site https://code.visualstudio.com/download e instale a última versão conforme o seu sistema operacional. Após a instalação, você precisará adicionar o **pacote de extensões da Salesforce**. Procure por `Salesforce Extension Pack` e realize sua instalação. Após essa etapa, você estará pronto para começar a desenvolver profissionalmente em Salesforce APEX.

Criando sua Org de Developer

Para desenvolver em Salesforce, podemos criar um ambiente, sem custo, que chamamos de **Org de Developer** (*Developer Org*). Essa Org fornece os mesmos recursos de um ambiente produtivo, porém com um volume de armazenamento de somente 250MB. Para isso, acesse o site https://developer.salesforce.com/signup e preencha o formulário.

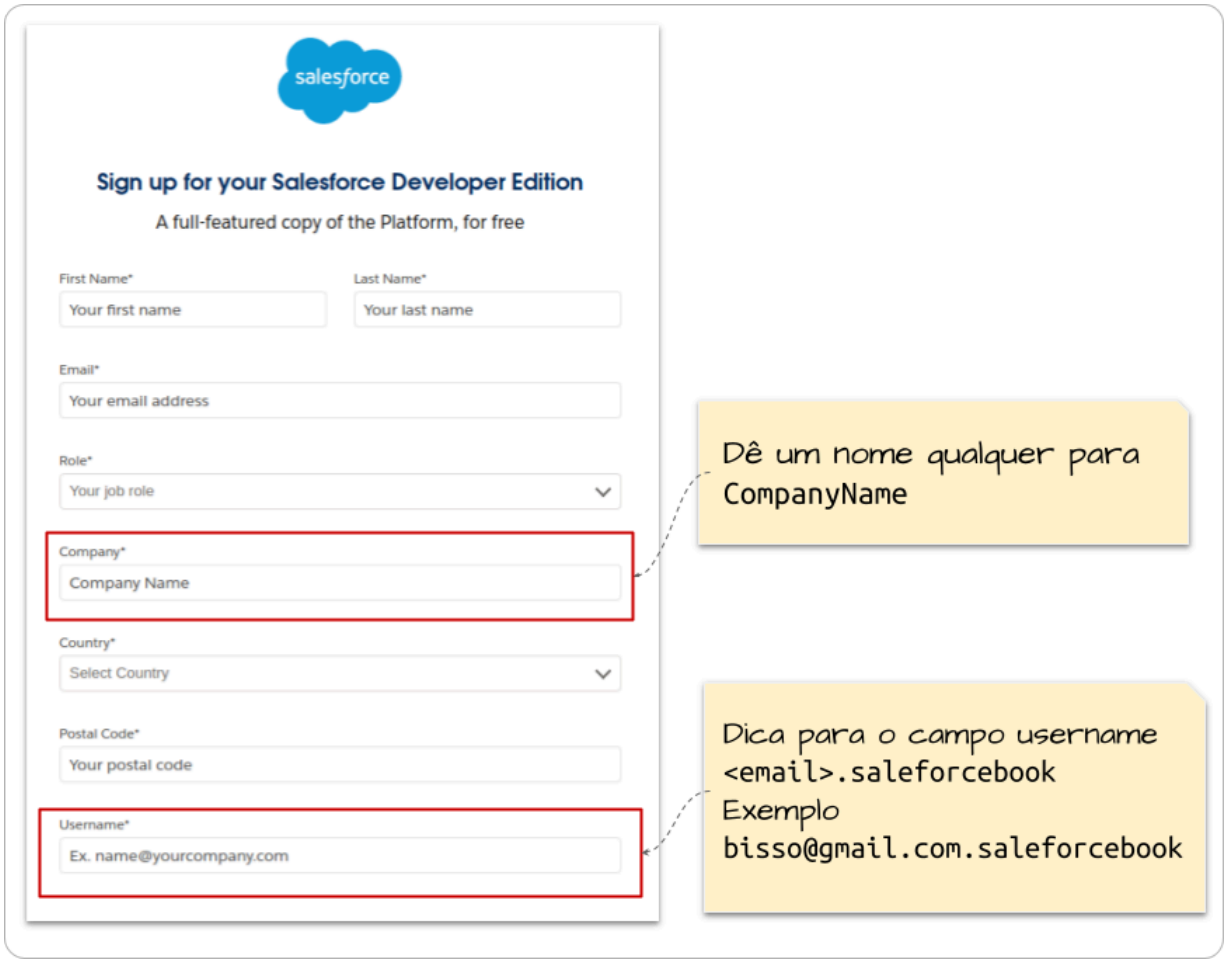


Figura 1.2: Dica para o form de Sign Up

Você vai receber um e-mail da Salesforce com o título "*Bem-vindo(a) ao Salesforce: Verifique sua conta*". Nele, clique em *Verificar Conta* e crie uma senha para sua Org.

Criando o seu projeto Salesforce no VSCode

Agora é a hora da verdade! Este é o começo da nossa jornada para desenvolver dentro da plataforma Salesforce. Agora que está tudo configurado, vamos criar o projeto conectando o VSCode com a sua **Org de Developer** recém-criada. Para isso, abra seu VSCode, acione a paleta de comandos com `Ctrl+Shift+P` e, em seguida, pesquise pela opção `SFDX: Create Project`. Será apresentada a janela de seleção de projeto, então selecione a opção `Standard Project`.

Após a definição do tipo do projeto, você deverá informar o nome do projeto, como `salesforce-book-apex-dev`, e informar a pasta onde deseja que o projeto seja criado, conforme mostra a imagem a seguir:

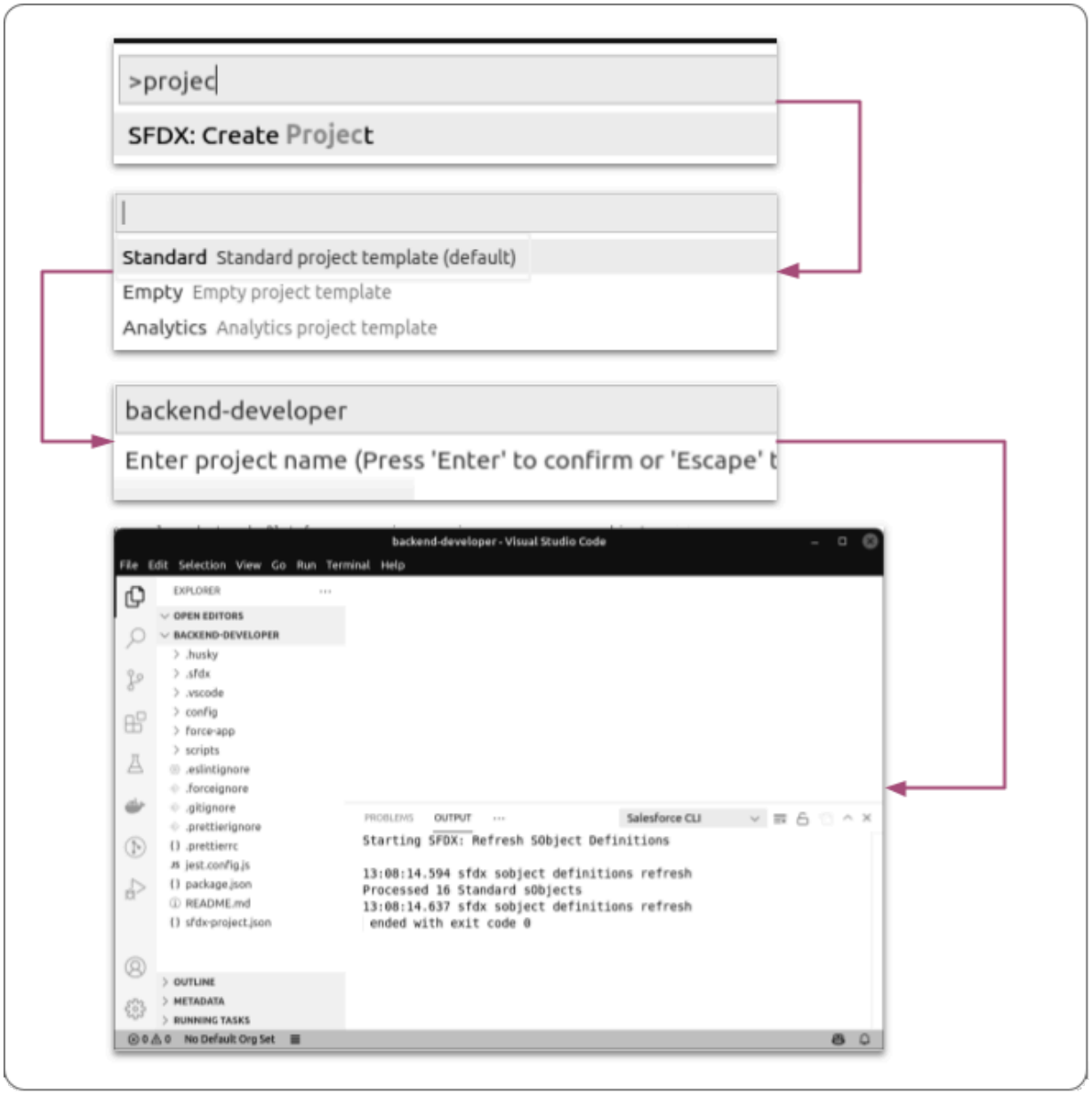


Figura 1.3: Criando projeto no VSCode

Após essa etapa, você precisa se autenticar na Org. Na paleta de comandos, pesquise e selecione a opção `SFDX: Authorize an Org` e, depois, `Project Default`. Informe um "apelido" para a sua Org, como

`salesforce-book-apex-dev`, por exemplo. Com isso, o VSCode vai abrir o seu *browser* para que você possa realizar a autenticação na **Org de Developer** recém-criada. Após a autenticação, você estará pronto(a) para desenvolver em Salesforce APEX.

Capítulo 2
Introdução a Orientação a Objetos

> "*Primeiro aprender a ficar em pé, depois aprender a voar. Lei da natureza.*" — Senhor Miyagi (*Karate Kid*)

2.1 Primeiros passos com a linguagem APEX

Este capítulo será como uma receita de bolo, justamente para que você aprenda a utilizar o VSCode junto do APEX. Para que você absorva melhor o conteúdo deste livro, recomendo que implemente todos os códigos contidos nele; por isso, é superimportante que você tenha o seu ambiente de desenvolvimento configurado, descrito no capítulo anterior.

Vamos ver o primeiro exemplo de uma classe em APEX. Nela, teremos alguns mistérios que ainda ficarão em aberto, mas tenha calma, tudo será esclarecido no passar dos capítulos. Nossa primeira classe tem a responsabilidade de representar uma `NotaFiscal` e imprimir uma mensagem no console.

```
public class NotaFiscal {

    public void imprimeNota (Decimal valor) {

        System.debug('Nota Fiscal de Compra : 2023233');

    }

}
```

Vamos ver agora o que cada parte desse código representa. Começaremos com a representação da classe `NotaFiscal`.

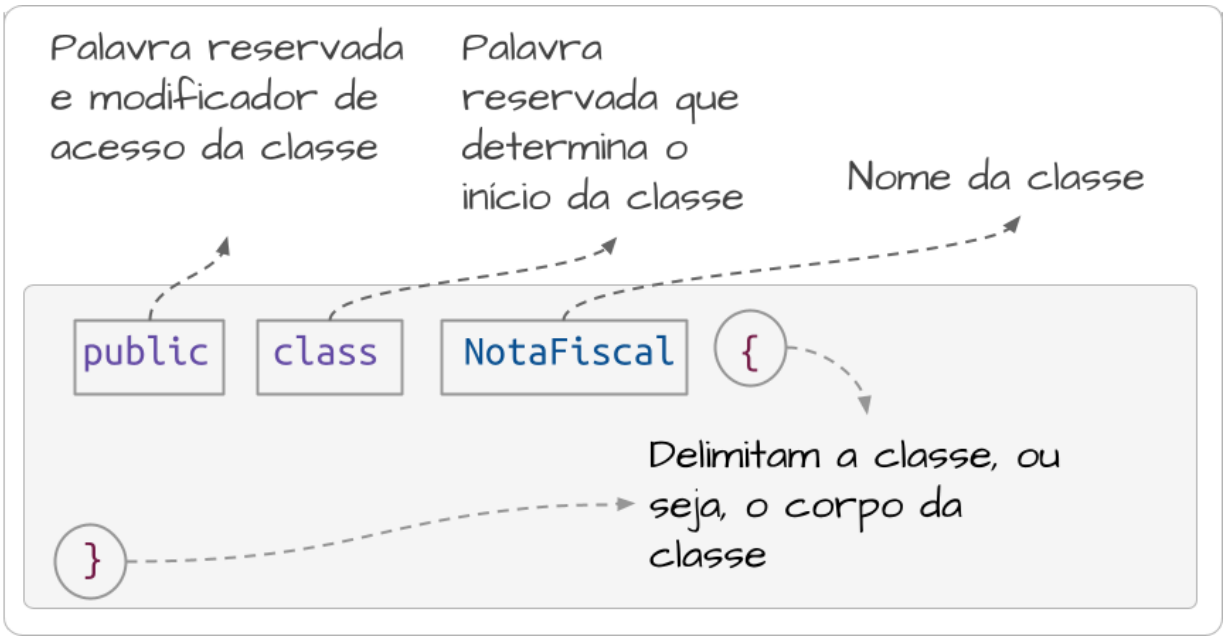


Figura 2.1: Anatomia de uma classe

- **`public`** : palavra reservada e o modificador de acesso da classe. Indica que ela está visível para outras classes;
- **`class`** : também uma palavra reservada e indica que se trata de uma classe;
- **`NotaFiscal`** : representa o *nome* da classe (já, já falaremos mais sobre padrões de nomenclatura);
- `{ ... }` : delimita o *escopo* da classe, também chamado de *corpo da classe*. Tudo que pertence à classe deverá estar dentro desse bloco de código.

Agora vamos ver o método `imprimeNota` e a sua anatomia.

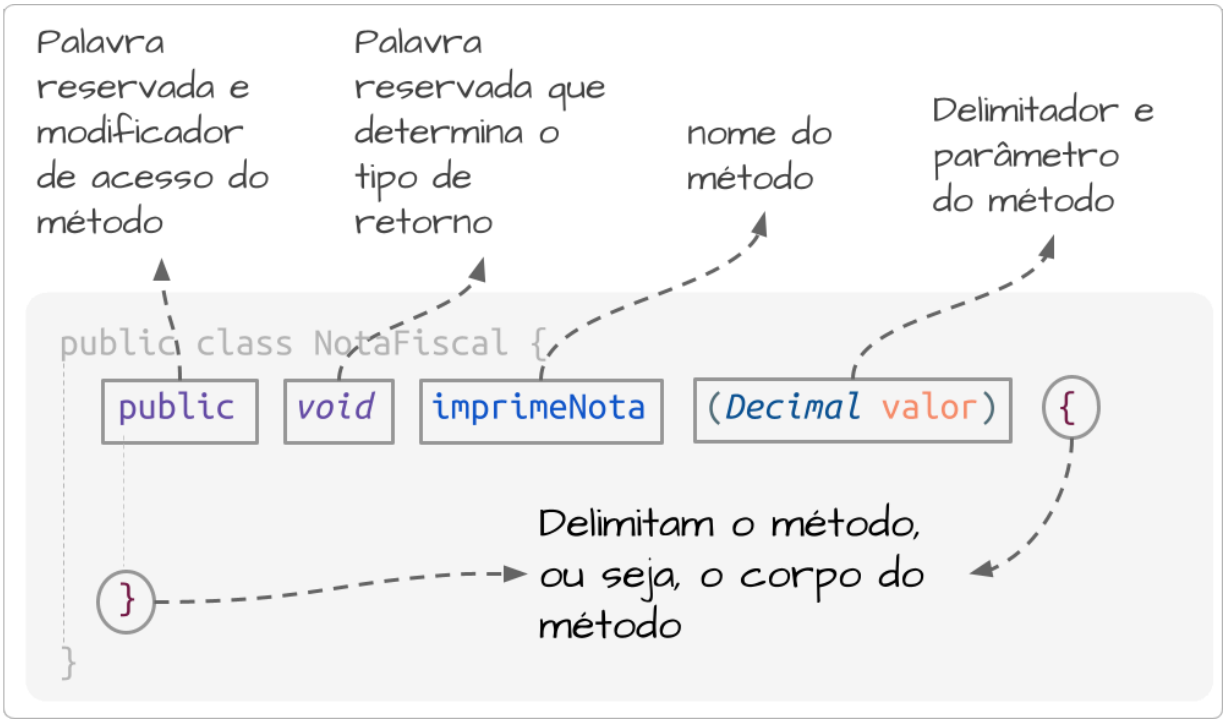


Figura 2.2: Anatomia de um método

- `public` : como no modificador de acesso utilizado para classe, ele delimita a visibilidade do método, ou seja, indica que o método pode ser acessado por outras classes. Veremos mais detalhes sobre visibilidade mais adiante;
- `void` : outra palavra reservada, que representa o tipo de retorno do método; no caso, representa retorno vazio;
- `imprimeNota` : representa o *nome* do método (já, já falaremos mais sobre esses padrões também);
- `( ... )` : delimita os parâmetros do método; eles são obrigatórios, contendo parâmetros ou não;
- `Decimal valor` : argumento ou parâmetro do método.
- `{ ... }` : delimita o *escopo* do método, também chamado de *corpo do método*. Tudo que pertence a ele deve estar dentro desse bloco de código;
- `System.debug('Nota Fiscal de Compra : 2023233');` : é um comando que será executado pelo método. Todo comando deve ser finalizado por ponto e vírgula (`;`).

Executando a nossa primeira classe

Para executar essa classe, crie uma nova aba no seu **VSCode** (`CTRL + N`) e digite o trecho a seguir:

```
NotaFiscal notaFiscal = new NotaFiscal();
notaFiscal.imprimeNota (1231.00);
```

Agora, para executá-la, vamos utilizar o recurso de execução de código anônimo através do comando `SFDX:Execute Anonymous Apex with Editor Contents` . Faça os passos a seguir e visualize o resultado no console:

1. Selecione todas as linhas;
2. Pressione `Ctrl+Shift+P` , digite `execute` e selecione `SFDX:Execute Anonymous Apex with Editor Contents` .

Acompanhe o resultado no console do VSCode. O *debug* é o seu melhor amigo!

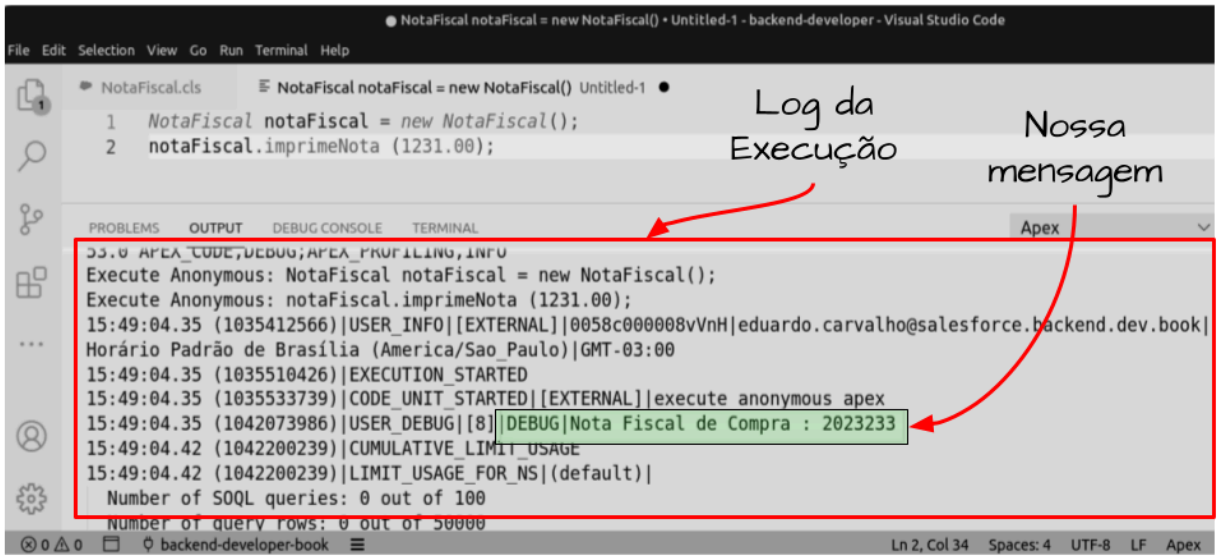


Figura 2.3: Log de execução

Observe o trecho destacado na imagem anterior, o resultado do comando `System.debug('Nota Fiscal de Compra : 2023233');` , que tem o

propósito de imprimir no console a mensagem `Nota Fiscal de Compra : 2023233 .`

Agora você já sabe como executar uma classe em APEX. Use e abuse desse recurso para executar partes de um código.

> **#se-liga-na-info** 😎
>
> A linguagem *APEX* realiza somente uma pré-validação local, mas todo conteúdo é compilado e salvo na **Org Salesforce**, ou seja, você não tem um *byte-code* local como em Java. Essa distinção é muito importante para entender o ciclo de vida de projetos Salesforce.

Estrutura de projeto Salesforce

Logo na introdução do livro menciono como criar o projeto em Salesforce utilizando o VSCode junto com o `SFDX` (*Salesforce Developer Experience*). Através deles, podemos desenvolver de forma profissional e rápida.

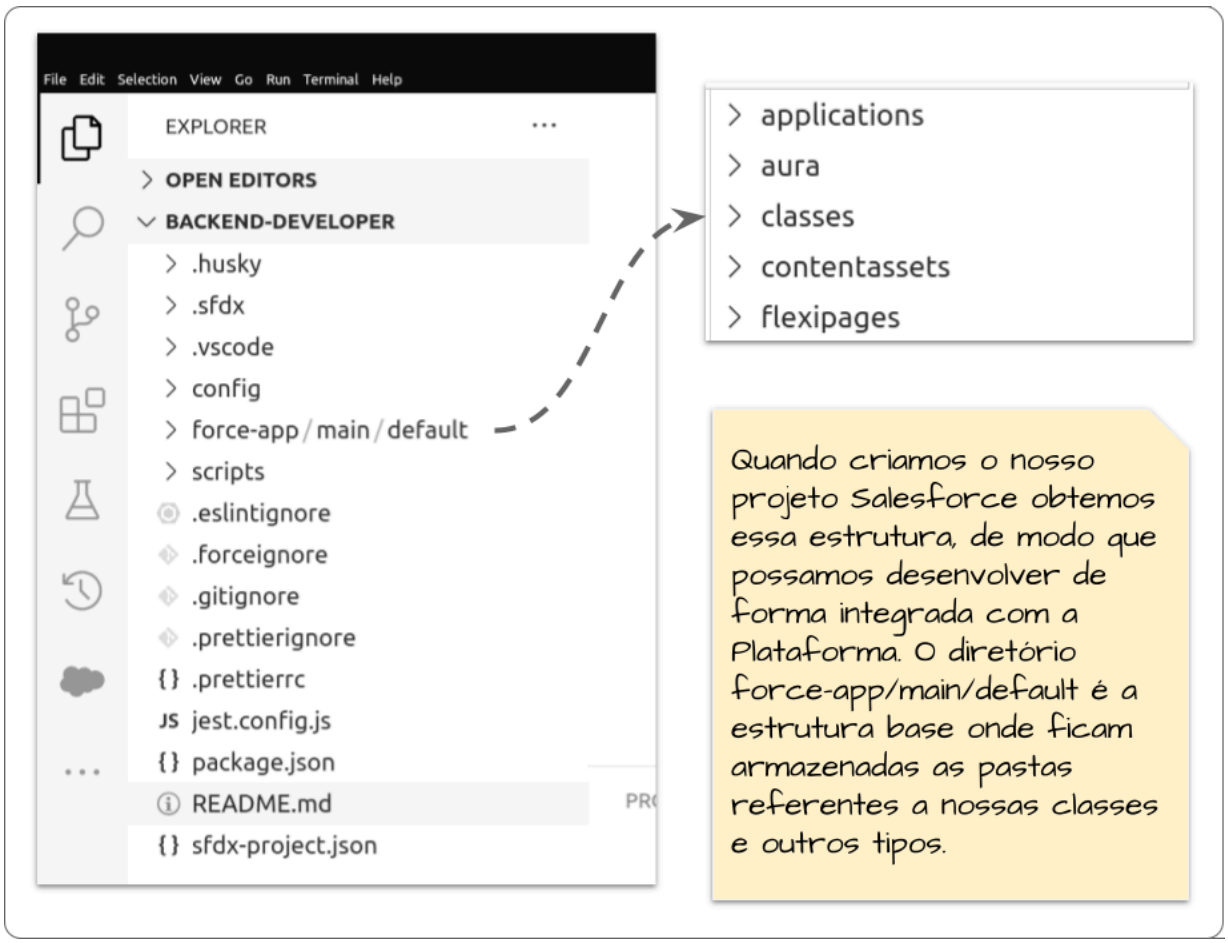


Figura 2.5: Estrutura de projeto APEX

Quando criamos a nossa classe `NotaFiscal`, o VSCode cria dois arquivos no projeto:

- **`classes/NotaFiscal.cls`**: arquivo que representa a classe propriamente dita;
- **`classes/NotaFiscal.cls-meta.xml`**: arquivo que representa o metadado da classe. Nele, estarão definidas informações como qual versão de API do Salesforce será utilizada e se a classe está ativa ou não.

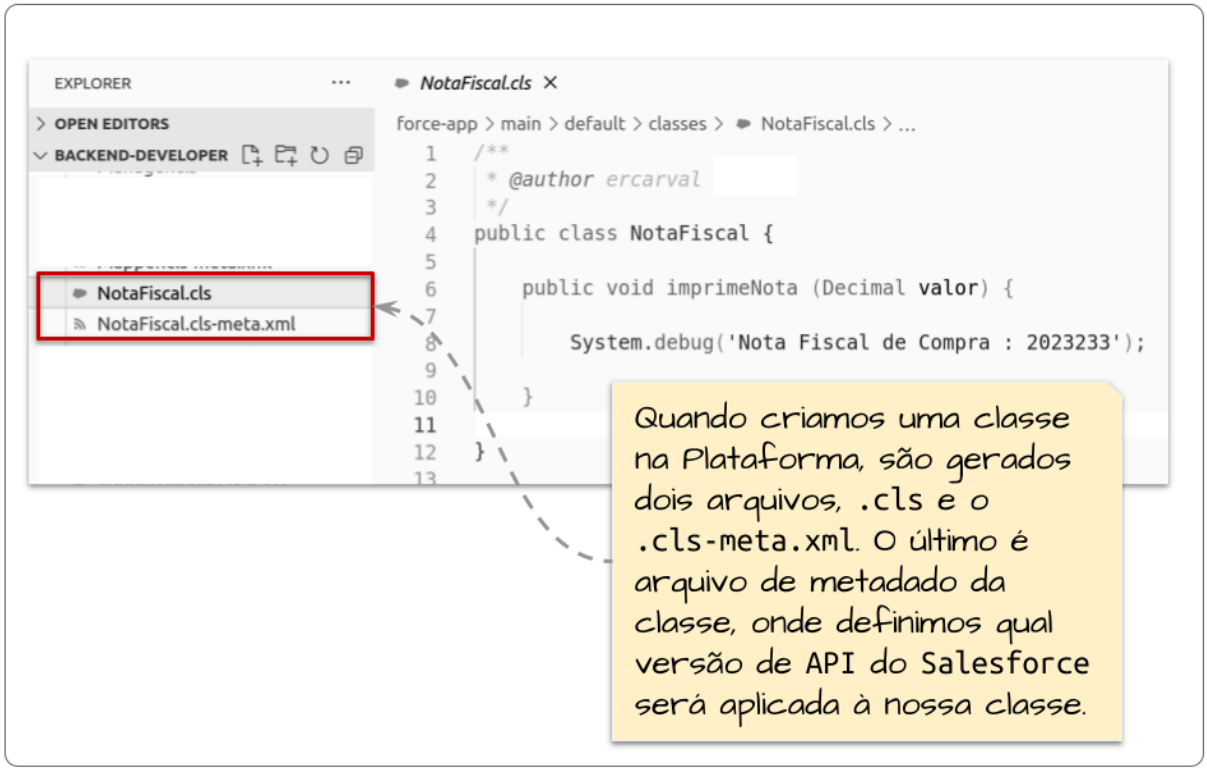


Figura 2.6: Estrutura de arquivos criados para uma classe

Entendendo um pouco mais dos metadados

Um dos pilares da plataforma Salesforce é a maneira como ela gerencia a versão de API de tudo que roda em seu ambiente. Isso é determinado pelos **metadados**, que determinam qual versão de API será utilizada para execução desse código, objeto, fluxo e centenas de outros tipos. Mas o que é essa versão de API?

A plataforma faz três atualizações ao ano e, a cada nova atualização, a versão de API é incrementada, identificando quais serão os recursos utilizados pelo código em questão. Em 2023 foi aplicada a versão de API `59.0`, em 2024 chegaremos até a `61.0`, e assim consecutivamente. Veja a seguir um exemplo do XML do metadado de uma classe:

```
<?xml version="1.0" encoding="UTF-8"?>
<ApexClass xmlns="http://soap.sforce.com/2006/04/metadata">
    <apiVersion>58.0</apiVersion>
```

```
    <status>Active</status>
</ApexClass>
```

Atualmente, a plataforma fornece compatibilidade reversa com versões bem antigas, sendo que algumas já estão sendo descontinuadas, como a versões `21.0` a `30.0`, que serão desativadas em 2023. Com isso, os clientes têm tempo hábil de atualizar suas aplicações para a versão mais recente.

2.2 Boas-vindas ao mundo dos Objetos

Agora que você criou sua primeira classe, vamos entrar de fato nesse tão falado mundo da Programação Orientada a Objetos! Mas antes observe o seu **celular**: quais são as suas **características**? Provavelmente você vai identificar parte dos itens a seguir: tamanho, cor, peso, marca, modelo, fabricante.

Se você o desmontasse, possivelmente acharia mais coisas, como microcontroladores, sensores e outros componentes eletrônicos. Então seu celular não é uma única peça; na verdade, ele é composto de vários outros componentes, e cada um deles também são compostos de outros, e todos possuem um conjunto de **características e comportamentos**.

Agora olhe ao seu redor e pense em outro objeto: a televisão, o computador, o aparelho de som, a batedeira ou qualquer outro. Todos possuem suas características e comportamentos! **Orientação a Objetos** nada mais é do que **uma forma de representar essas "coisas" em software**, que em ***APEX*** nós representamos através de uma **classe** ou um conjunto delas.

Como vimos, podemos **representar o mundo real através de software** (usando a Orientação a Objetos) e, para representar o celular, podemos criar uma classe também chamada de `Celular`:

```
public class Celular {

}
```

2.3 Determine as características de uma classe

Para representar as características de uma classe, você deve declarar **variáveis** no corpo da classe. Estas também são chamadas de *variáveis de instância*, ou *atributos de instância*. Veja o exemplo a seguir:

```
/**
 * Classe responsável por representar o aparelho de Celular
 * @author: eduardo.bisso
 */
public class Celular {

    public Integer numeroNucleosProcessador;

    public String marca;

    public String fabricante;

}
```

#se-liga-na-info 😎

Alguns pontos ainda ficarão sem respostas, como o modificador `public` e o `with sharing`, se você já estiver codificando em conjunto no VSCode. O último nem foi mencionado ainda nos exemplos :). Todos esses pontos serão vistos mais à frente.

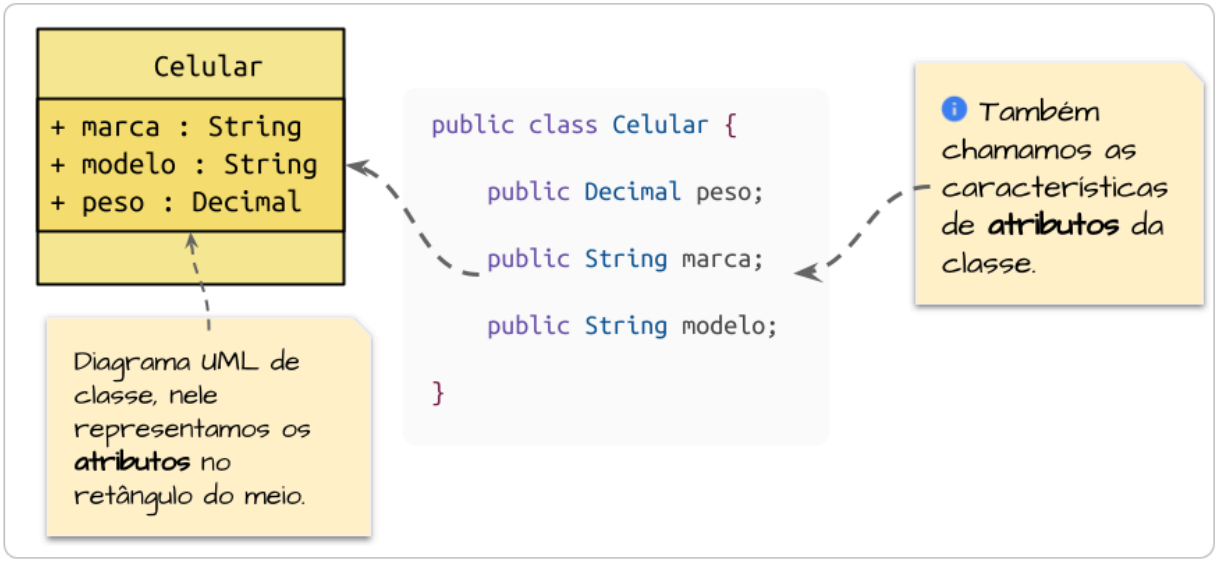


Figura 2.8: Representação em UML da nossa classe

Agora que você já viu como é fácil criar uma classe, vamos executá-la e ver como será a sua utilização:

```
Celular motorolaXPTO = new Celular();

motorolaXPTO.numeroNucleosProcessador = 8;
motorolaXPTO.marca = 'Moto XPTO';
motorolaXPTO.fabricante = 'Motorola';

System.debug ( 'Motorola XPTO ' + motorolaXPTO);

Celular iPhoneXXX = new Celular();

iPhoneXXX.numeroNucleosProcessador = 8;
iPhoneXXX.marca = 'iPhone';
iPhoneXXX.fabricante = 'Apple';

System.debug ( 'iPhoneXXX  ' + iPhoneXXX);
```

Para executar o código anterior, utilize novamente o comando `SFDX`:

```
Execute Anonymous Apex with Editor Contents .
```

Entendendo o código anterior

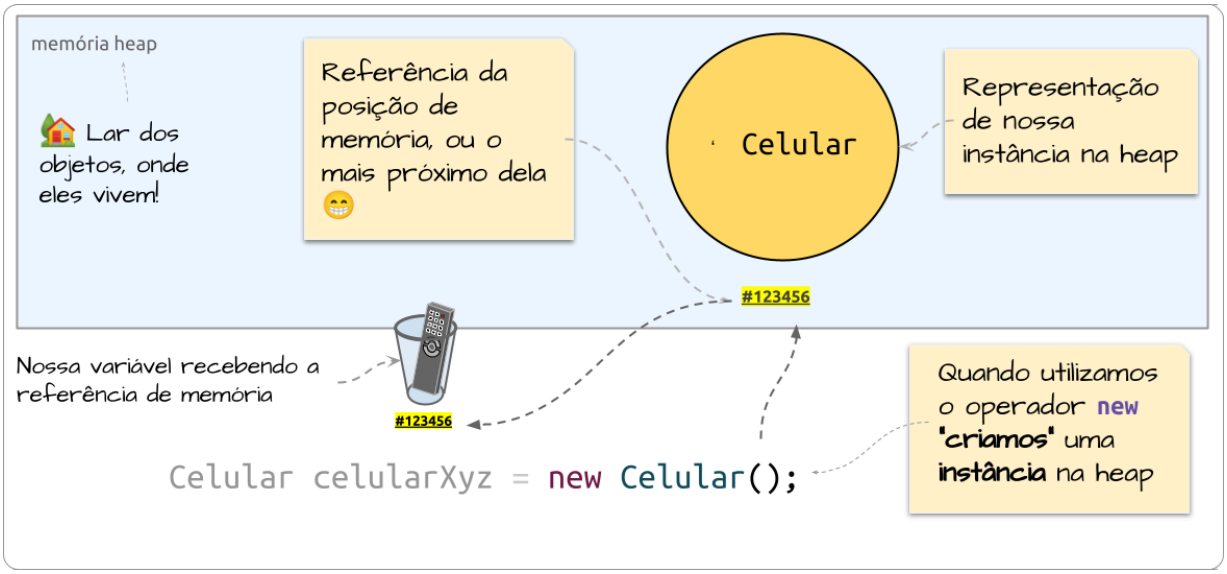


Figura 2.9: Objeto na memória heap

Quando utilizamos a palavra reservada `new <TipoClasse>()`, estamos criando uma **instância** daquela classe na *memória heap*. Então, para utilizar a classe `Celular`, é necessário criar uma instância dela.

Veja a sintaxe de uma variável e a criação de sua instância:

```
<Tipo> <nomeDaVariável> [ = new <Tipo> () ];
```

Agora, aplicando ao nosso exemplo, temos:

```
Celular celularXyz = new Celular();
```

Agora que vimos como uma **instância** é criada e representada na memória *heap*, vamos ver como determinamos suas características e sua representação na *heap*:

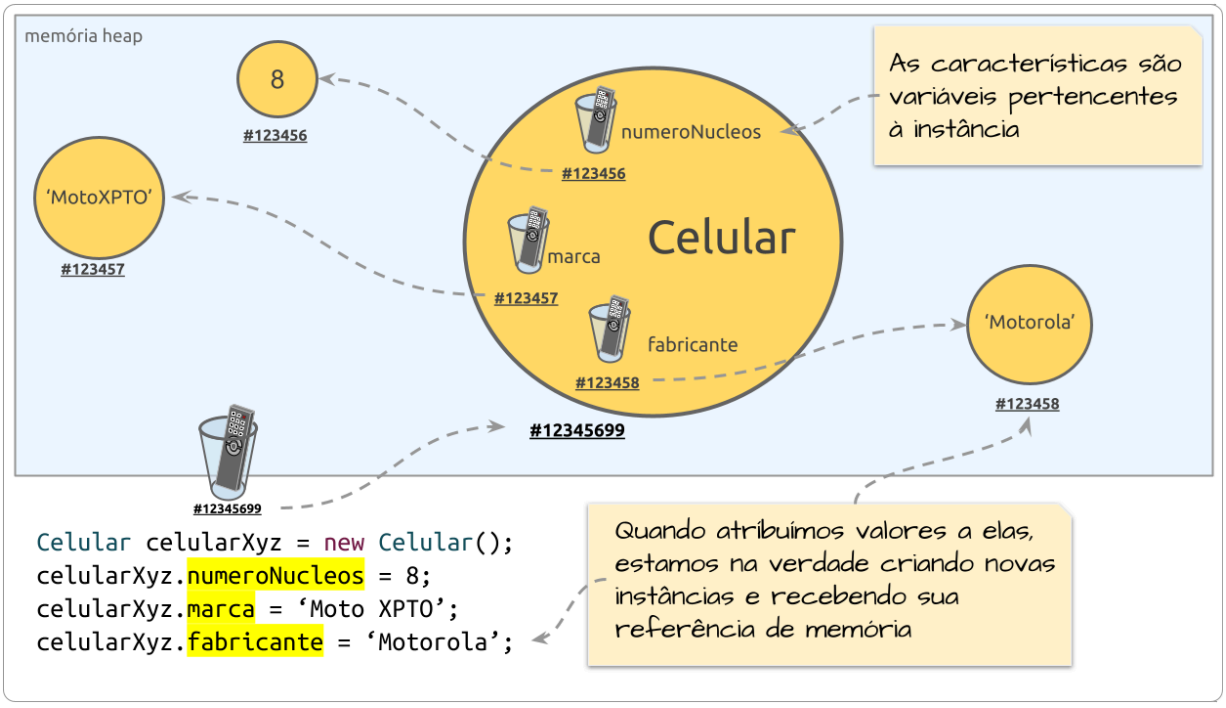


Figura 2.10: Variáveis de instância na memória heap

Perceba que estamos utilizando o ponto (.) para acessar as variáveis definidas na classe `Celular`. Através dele, acessamos os membros de nossa classe.

Note a representação das variáveis: pense nelas sempre como copos, que recebem uma referência de memória. Quando instanciamos os objetos e determinamos suas características, estamos criando um espaço de memória que concentra as informações para aquele objeto/instância em questão, e ela, instância, contém um conjunto de outros copos, que são suas variáveis de instância.

#se-liga-na-info 😎

Convenção é comunicação

Todo desenvolvimento de código precisa ter uma convenção bem estabelecida, para que todos possam entender o que foi desenvolvido, sem aumentar o esforço cognitivo. Este livro é baseado na convenção de código Java, a qual será abordada durante todo o decorrer do livro.

Convenção de classe

Para a nomenclatura de classes, utilizamos o padrão **P**ascal**C**ase, que determina que a primeira letra da palavra deve ser em **maiúscula** e o restante, em **minúscula**. Se houver composição de palavras, a primeira letra da palavra subsequente deve sem em **maiúscula**. **Não** usamos *underline* (_) para separar as palavras. Exemplo: **B**ack**C**olor. Veja o código a seguir:

```
public class Customer {

}

public class NotaFiscal {

}

public class AdvancedShippingNotes {

}
```

Convenção de variáveis

Para a nomenclatura de variáveis, utilizamos o padrão **c**amel**C**ase, que determina que a primeira a palavra deve ser em minúscula; se houver composição de palavras, a primeira letra da palavra subsequente deve ser em maiúscula e o restante, em minúscula. Exemplos: **e**ndereco**E**ntrega, **n**ota**F**iscal. Veja o código a seguir:

```
String enderecoEntrega;

NotaFiscal notaFiscal;

String nome;

DateTime dataCriacao;
```

2.4 Determinando comportamento de uma classe

O que são os comportamentos de uma classe? Nada mais são que as ações (operações) realizadas pelos nossos objetos. Olhando novamente a nossa classe `Celular`, provavelmente você poderia listar alguns destes comportamentos:

- Ligar;
- Desligar;
- Tirar foto;
- Enviar mensagem;
- Instalar aplicativo;
- Visualizar galeria de fotos;
- Adicionar contato.

Como determinamos esses comportamentos em uma classe? Através da declaração de métodos no corpo dela:

```
/**
 * @author: eduardo.bisso
```

```
 */
public class Celular {

    // atributos comentados

    public void discar (String numeroDestino) {

        System.debug ('Ligando para '
                + numeroDestino
                + '\n De : ' + numero
                + '\n Marca/Modelo : '
                + marca + '/' + modelo);
    }


    public void enviarMensagem (String numero
                                , String mensagem) {

        System.debug ('[ ' + numero + ' ]\n'
                    + 'Mensagem : ' + mensagem );
    }

}
```

Os métodos do código anterior pertencem à instância da classe; ou seja, para acessá-los, é necessário que um objeto seja criado e instanciado para que possa ser chamado. Vamos ver graficamente como seria isso.

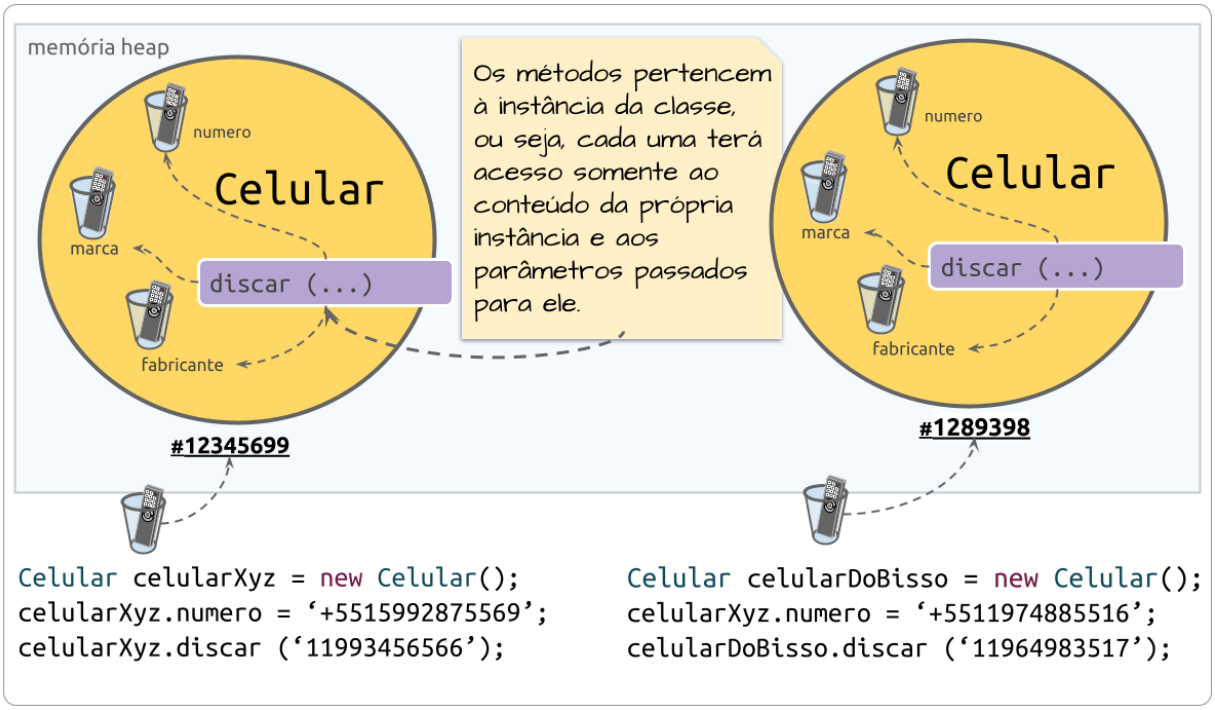


Figura 2.12: Métodos de instância e a memória heap

Perceba que cada instância terá sua "versão" de método. Cada uma acessa o seu método em si através do ponto (.), assim como acessamos os atributos (variáveis pertencentes à instância).

Agora que você já viu como os métodos são definidos e como eles ficam na memória *heap*, vamos repassar a sua **sintaxe**:

```
[modificadores] <TipoDeRetorno> <nomeDoMétodo> ( [parâmetros] ) {

    //corpo do método

}
```

- **`modificadores`** : são opcionais e determinam a visibilidade do método, podendo haver mais de um. Em nosso exemplo do `Celular` , está representado pelo `public` ;
- **`TipoDeRetorno`** : determina qual o tipo de retorno do método. Em nosso exemplo do `Celular` , é representado pela palavra `void` , que significa *ausência de retorno* ou *retorno vazio*;

- **`nomeDoMétodo`** : como comentado anteriormente, deve ser escrito seguindo a convenção `camelCase` . Recomendo utilizar um verbo como parte do nome, dado que ele representa uma ação;
- `( ... )` : delimita os parâmetros do método. Eles são obrigatórios, contendo parâmetros ou não;
- **`parâmetros`** : são opcionais, e pode haver mais um. Eles são variáveis que são passadas para utilização do método;
- `{ ... }` : delimita o *escopo* do método, também chamado de *corpo do método*. Tudo que pertence a ele deve estar dentro desse bloco de código.

#se-liga-na-info 😎

Convenção - Métodos

Para os métodos, utilizamos o padrão **c**amel**C**ase, a mesma convenção que aplicamos para as variáveis. Exemplo: **i**mprime**N**ota.

```
public class Customer {

    public Date getLastModifiedDate () {
        // ....
    }

}

public class NotaFiscal {

    public void imprimeNota () {
        // ....
    }

}
```

Padrão de formatação

A formatação de código está diretamente ligada a nossas convenções de codificação, o *Code Style*. Devemos ter **extrema atenção a esse ponto**, pois ele demonstra quão preocupados estamos com a organização de nosso código. Se deixamos uma classe desformatada, estamos indicando que não temos muito apreço à organização, o que pode significar uma tendência de que outros pontos, como a lógica, também estejam desorganizados ou mal estruturados.

Como estamos utilizando o padrão Java, a primeira chave `{` do bloco de código deve estar alinhada à direita do comando e a última chave `}` deve estar alinhada com a letra do comando que iniciou o bloco, *indentando* o conteúdo do bloco com `4` espaços. Não leve em consideração o conteúdo do algoritmo a seguir, mas, sim, a sua formatação:

```
public class Pedido {

    String tipoPedido;

    public Decimal calculaValorTaxa () {

        if (tipoPedido == 'SERVIÇO') {
            return 0.2;
        } else if ( tipoPedido == 'VENDA PADRÃO' ) {
            return 0.1;
        }

        return 0;

    }

}
```

2.5 Ciclo de vida dos objetos

Agora que vimos com mais detalhes como os objetos são criados/instanciados, vamos colocar uma lupa nessa etapa para entender como é o ciclo de vida de uma instância dentro da memória *heap*.

Construtores

Os **construtores** são os responsáveis por criar/instanciar os objetos na memória *heap*. Quando utilizamos o trecho de código `new Celular();`, estamos utilizando justamente o construtor da classe `Celular`. Observe novamente os trechos de código:

```
public class Celular {
    // atributos omitidos
}

// criando uma instância de Celular.

Celular celularXyz = new Celular();
```

Quando utilizamos o `new Celular();` estamos executando o *construtor* da classe `Celular`, mas você deve estar pensando: onde esse construtor está escrito? O detalhe não comentado anteriormente é que toda classe possui um construtor, porém, quando não o declaramos explicitamente, o compilador nos dá um de presente. É como se tivéssemos escrito o nosso código da seguinte forma:

```
public class Celular {
    // atributos omitidos

    public Celular () {
        // corpo do construtor
    }

}
```

Note que agora apareceu um novo trecho de código, `public Celular ()`. Ele também é chamado de **construtor padrão** (*default*), pois

não possui argumentos/parâmetros. Ele é semelhante a um método, porém ele deve ter o mesmo nome da classe e não possuir um retorno. Veja a seguir a sua sintaxe:

```
[modificador] <NomeDaClasse> ( [parâmetros...] ) {

    // corpo do construtor

}
```

Geralmente o construtor é utilizado para o fluxo de inicialização dos atributos/variáveis de instância da classe. Então você deve utilizá-lo para a definição de valores padrões para suas variáveis de instância da classe, como no código apresentado a seguir, onde desejamos que nossas instâncias de `Celular` sempre sejam criadas como *desligadas*, com `ligado = false;` :

```
public class Celular {

    public Boolean ligado;

    public Celular () {

        ligado = false;

    }

}
```

#se-liga-na-info 😎

Todos os atributos de instância são inicializados com `null` .

Veja graficamente o ciclo de criação de uma classe:

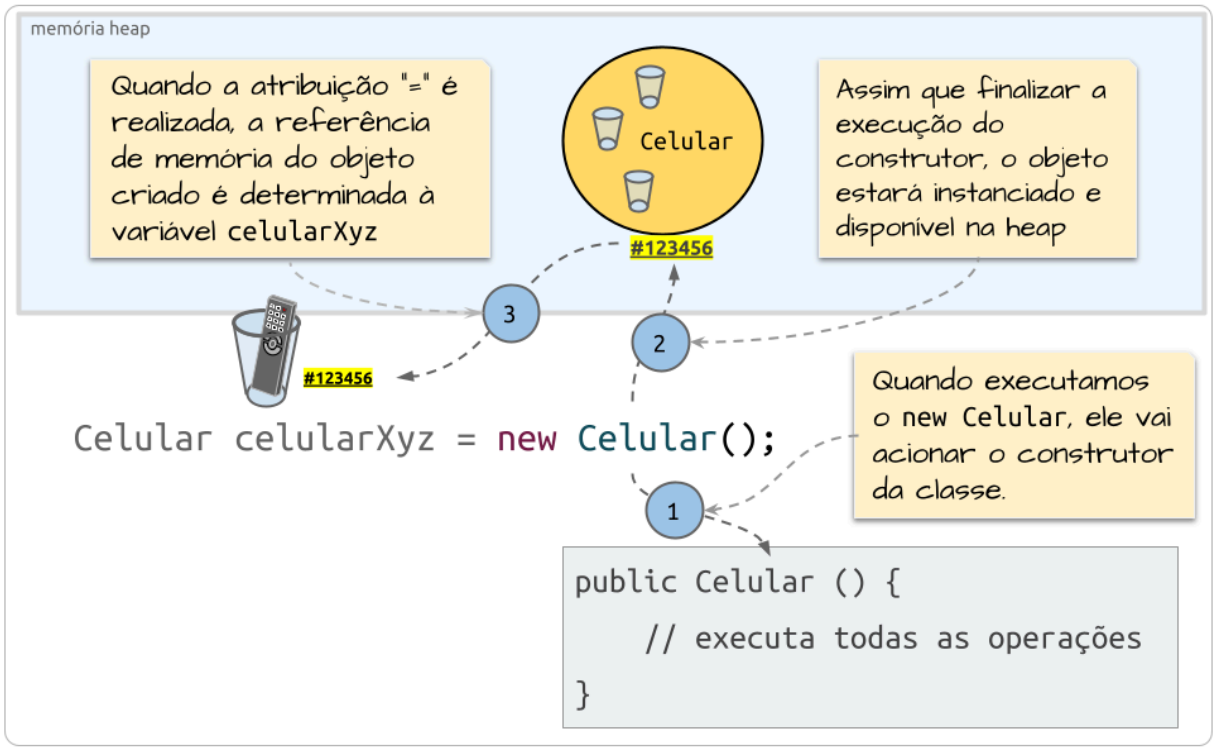


Figura 2.15: Ciclo de vida de inicialização de uma classe

2.6 O que é este tal de this?

Antes de contar o que é esse tal de `this`, observe o código a seguir. Note que estamos utilizando o *construtor* para determinar o número do `Celular` quando ele for instanciado:

```
/**
 * @author: eduardo.bisso
 */
public class Celular {

    Long numero;

    public Celular (Long numero) {
        numero = numero; // isso pode Arnaldo?
```

```
    }
}
```

Tem algo esquisito na atribuição `numero = numero;` ? Então vamos vê-la mais graficamente.

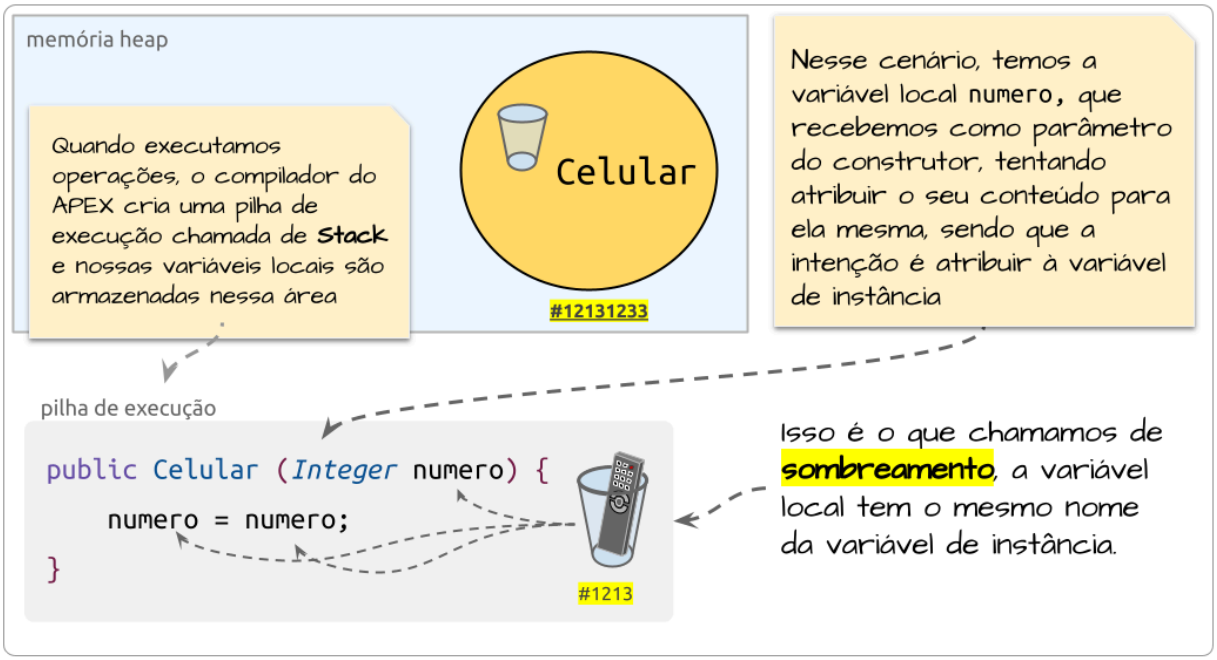


Figura 2.16: Sombreamento de variáveis

Como descrito na imagem, o sombreamento ocorre quando uma variável local possui o mesmo nome da variável de instância. Dessa forma, precisamos utilizar a palavra reservada `this` para "driblar" o sombreamento e fazer a atribuição corretamente. O `this` é a representação da própria instância, tipo "Edson" falando de "Pelé".

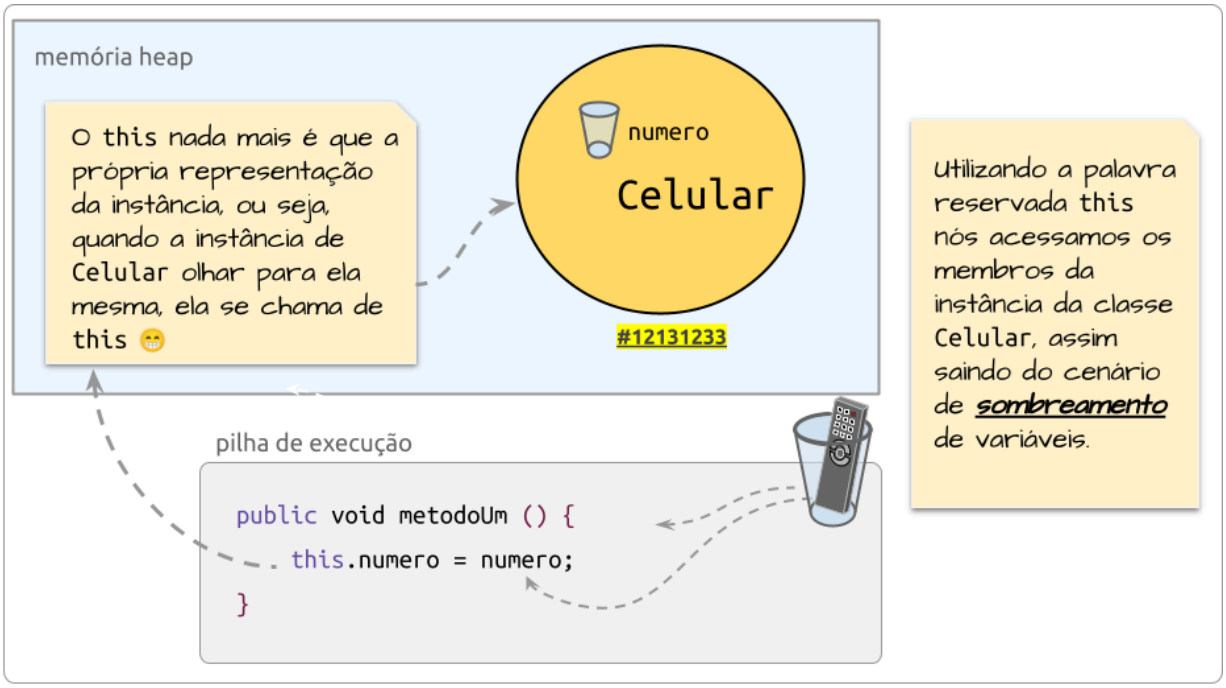


Figura 2.17: Utilizando o this para driblar o sombreamento

Utilizamos a palavra reservada `this` para acessar os membros da instância, portanto é recomendado que você utilize o `this` sempre que for fazer referência a algum membro (variável ou método) da própria instância. Essa prática fará com que o seu código fique mais legível, identificando se você está utilizando uma variável local ou uma de instância.

Fechando o ciclo de vida com o Garbage Collector

Nós vimos como é o "nascimento" dos objetos e, para fechar o ciclo de vida deles, vamos ver como um objeto "morre". A linguagem APEX, que é baseada na linguagem Java, utiliza um conceito de remoção de objetos que não são mais utilizados, chamado de *Garbage Collector* (coletor de lixo). Esse conceito está presente em diversas outras linguagens, como o próprio Java, C#, JavaScript, entre outras.

E como o *Garbage Collector* faz para identificar que um objeto pode ser coletado? Ele identifica os objetos que não possuem mais uma

referência, ou seja, quando não houver nenhuma variável "apontando" para aquela instância. O *Garbage Collector* marca os objetos em memória para sua remoção e depois os remove da memória.

No exemplo de código a seguir, vamos executar uma operação que envolverá a chamada de três métodos encadeados; em cada um, será criada uma variável local ao método:

```
void metodoUm () {

    Integer valorMetodoUm = 35;

    metodoDois (valorMetodoUm);

}

void metodoDois (Integer valor) {

    Integer novoValor = valor * 2;

    metodoTres ( novoValor );

}

void metodoTres (Integer valor) {

    String texto = 'Último método executado';
}
```

Não tem como falar do *Garbage Collector* sem entendermos o ciclo de execução dos métodos. Eles são empilhados no que chamamos de *stack* (pilha), e cada variável local fica alocada na sua área da pilha. Veja a imagem a seguir, que apresenta a execução do exemplo anterior como uma linha do tempo:

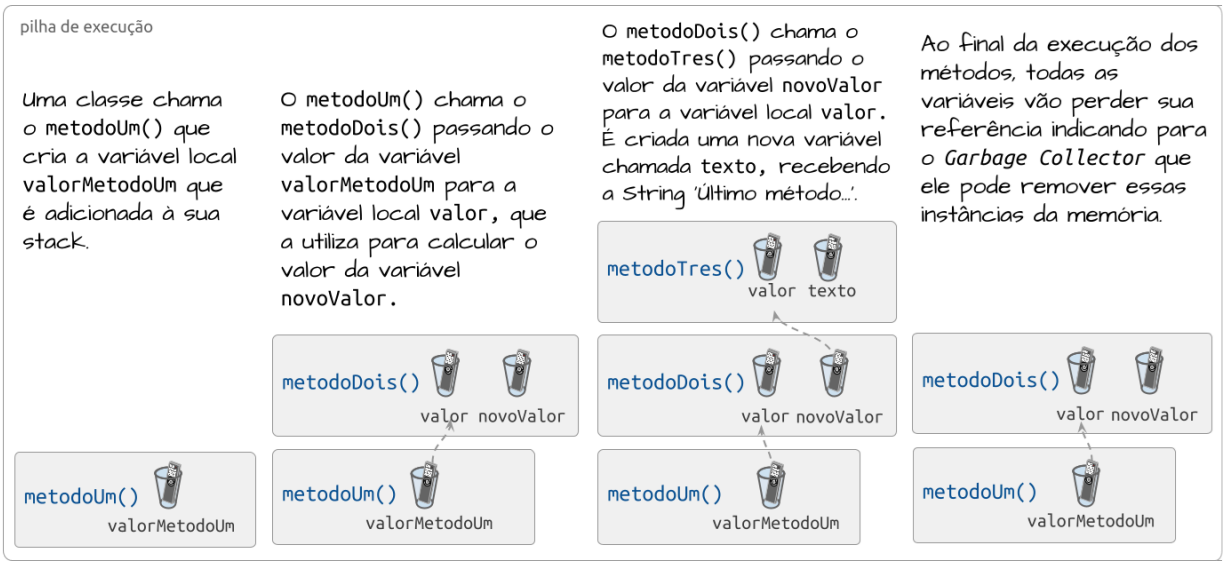


Figura 2.18: Stack trace

Conforme os métodos são executados, eles são removidos da *stack* e, com isso, suas variáveis, que foram instanciadas, vão perdendo a referência. Isso indica para o compilador que ela pode ser coletada pelo *Garbage Collector*.

Então, a *stack* é o lar dos métodos e variáveis locais, e a *heap* é o lar dos objetos e variáveis de instância. Vamos ver o exemplo agora criando instâncias de nossa classe `Celular` :

```
void metodoUm () {

    Integer numero = 122321321

    Celular celular = new Celular (numero);

    metodoDois (celular);

}

void metodoDois (Celular celularMetodoDois) {

    Celular novoCelular = new Celular (celularMetodoDois.numero);
```

}

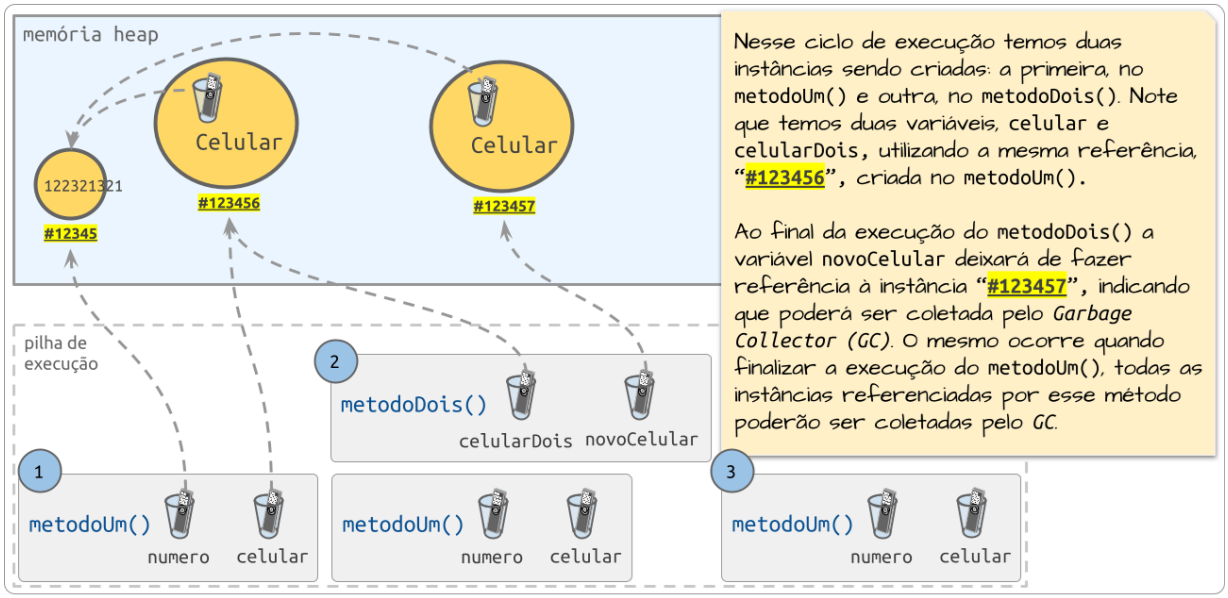


Figura 2.19: Ciclo de vida de objetos vs. stack trace

A imagem ilustra o ciclo de vida de uma instância e como ela passa a ser "marcada" para ser coletada pelo *Garbage Collector*.

#se-liga-na-info 😎

A Salesforce nos impõe um limite de somente 6 MB de memória *heap*, portanto temos que ter atenção em como implementamos nossas soluções. Tais limites nos forçam sempre a pensar na melhor solução para um determinado problema. No decorrer do livro, descreverei várias alternativas para que nossa implementação fique sempre elegante, respeitando os limites da plataforma.

2.7 Trabalhe com coleções de objetos

Coleções de objetos nada mais são que um *conjunto de dados*. Por exemplo, quando temos a necessidade de saber todos os *Leads* não convertidos, teremos um conjunto/coleção de objetos do tipo `Lead` com o `Status` diferente de "Convertido".

Existem várias formas de representar uma coleção, como uma *lista*, uma *pilha*, um *conjunto*, um *mapa* (ou *dicionário*). Veja a seguir como representar as coleções de objetos em APEX.

Arrays e listas

O modo mais simples de utilizar uma coleção é através de um **array** e/ou **lista** (`List`). Ele nada mais é que uma lista de objetos que tem as seguintes características:

- Mantém a ordem em que os elementos são adicionados;
- Acessa o conteúdo de forma indexada — posição `0` , `1` , `2` etc.;
- Pode armazenar elementos nulos;
- Pode conter elementos duplicados.

Veja a seguir a sintaxe de declaração de um array/lista.

Sintaxe de um array/lista

```
<Tipo[]> <nomeDaVariavel> [ = new <Tipo>[<tamanho>]];
```

Agora, um exemplo simples de declaração e adição de elementos no array:

```
Celular[] celulares = new Celular[10];

celulares.add ( new Celular ());
celulares.add ( new Celular ());
```

A linguagem APEX não distingue um array de uma lista. Quando utilizamos essa sintaxe, por baixo dos panos temos uma lista (`List`) em mãos, então recomendo que sempre utilize a sintaxe a seguir:

```
< List<Tipo> > <nomeDaVariavel> [ = new List<Tipo>() ];
```

#se-liga-na-info 😎

As coleções, de uma forma geral, utilizam a notação "diamante" (`<>`) em sua sintaxe. Essa notação faz com que somente elementos do tipo determinado entre `<>` possam ser adicionados à coleção.

Exemplo

```
List<Celular> celulares = new List<Celular>();

celulares.add ( new Celular ()); // só pode adicionar Celular
celulares.add ( new Celular ());
```

Note que, nesse exemplo, só estamos adicionando `Celular`, pois, ao definirmos o tipo da coleção como `List<Celular>`, indicamos que só será permitido esse tipo. Caso tente adicionar outro tipo, como `celulares.add ( new Customer ())`, isso geraria um erro de compilação.

Se você pudesse visualizar a `List` na *heap*, ela teria uma aparência semelhante à imagem a seguir:

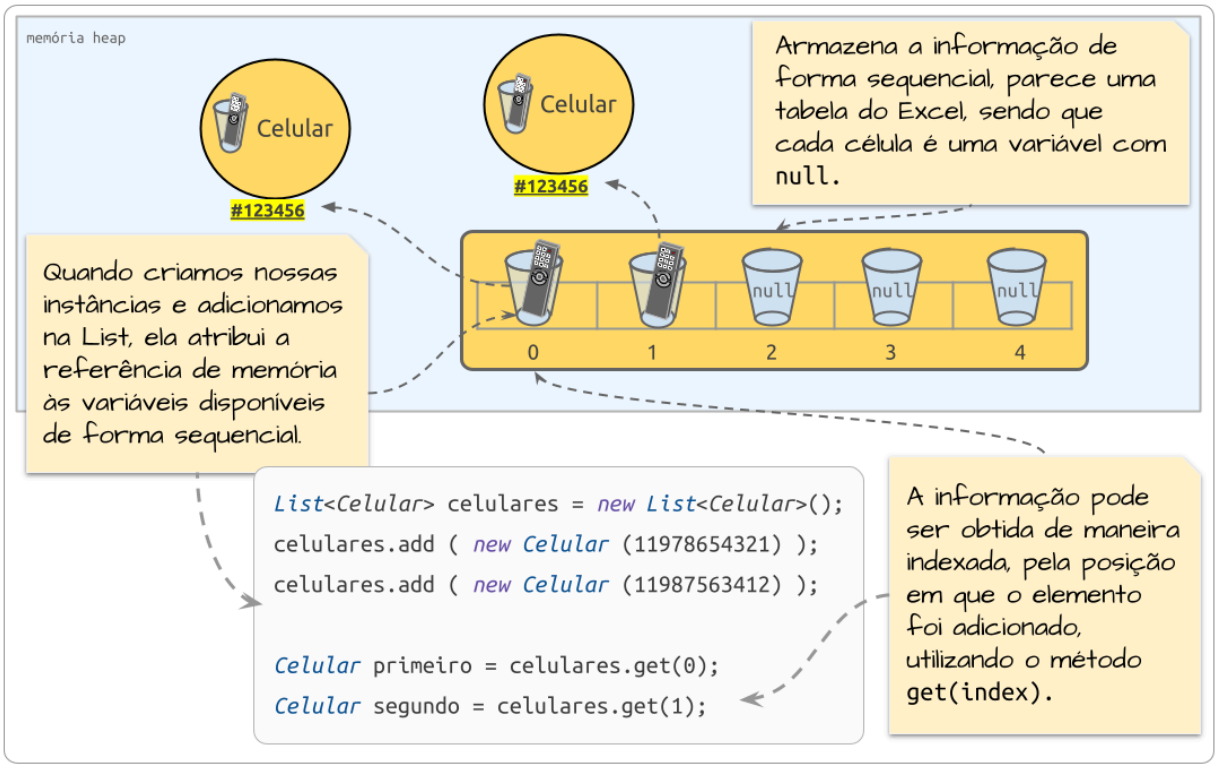


Figura 2.22: Olhando a coleção mais de perto

Conheça sua API

Todo(a) *dev* deve conhecer a API da linguagem na qual se está trabalhando. Então vamos ver alguns métodos fornecidos pela classe `List`.

- **`add (element)`** : adiciona um elemento à lista.

```
List<Celular> celulares = new List<Celular>();

celulares.add ( new Celular ());
```

- **`addAll (elements)`** : adiciona os elementos de uma outra lista na lista corrente.

```
List<Celular> motorolas = new List<Celular>();

motorolas.add (new Celular());
```

```
List<Celular> iphones = new List<Celular>();

iphones.add (new Celular());

List<Celular> celulares = new List<Celular>();
celulares.addAll ( motorolas );
celulares.addAll ( iphones );

System.debug ( celulares.size() );// Exibe 2 elementos;
```

- **`clear()`** : limpa o conteúdo da lista.

```
List<Celular> celulares = new List<Celular>();
celulares.add ( new Celular ());

System.debug ( celulares.size() );// Exibe 1 elemento;

celulares.clear();// limpa todo conteúdo da lista.

System.debug ( celulares.size() );// Exibe 0;

System.debug ( celulares.isEmpty() );// Exibe true;
```

- **`isEmpty()`** : indica se a lista está vazia, retornando `true` ou `false` .

```
List<Celular> celulares = new List<Celular>();

if (celulares.isEmpty()) {
   // execute alguma operação
}

System.debug ( celulares.size() );// exibe 0;
```

- **`get(index)`** : recupera elementos da lista de forma indexada, sempre iniciando por `0` .

```
Celular celularXyz = new Celular (11978654321L);
Celular celularBisso = new Celular (11987563412L);
```

```
List<Celular> celulares = new List<Celular>();
celulares.add ( celularXyz );
celulares.add ( celularBisso );

System.debug ( celulares.size() );// exibe 2;

Celular primeiro = celulares.get(0);
System.debug ( primeiro.numero ); // exibe 11978654321

Celular segundo = celulares.get(1);
System.debug ( segundo.numero ); // exibe 11987563412
```

Para ver os outros métodos de `List`, consulte a documentação da classe em: https://developer.salesforce.com/docs/atlas.en-us.apexref.meta/apexref/apex_methods_system_list.htm.

Set

A coleção `Set` traz uma implementação que **não permite elementos duplicados** e **não mantém a ordem** dos elementos na coleção.

```
Set<String> estados = new Set<String>();

estados.add ('SP');
estados.add ('RJ');
estados.add ('RJ'); // vai ignorar este elemento.

System.debug ( estados.size() ); // Exibe 2 elementos
```

Percebeu a semelhança? Isso acontece porque ela compartilha de parte da mesma implementação da `List` por baixo dos panos, e com isso os métodos `size`, `isEmpty`, `contains` também são fornecidos por ela. Geralmente utilizamos essa coleção com a intenção de não possuir elementos duplicados, mas lembre-se de que ela bagunçará a ordem dos elementos.

Para ver os outros métodos permitidos em `Set`, consulte a documentação da classe, em:

https://developer.salesforce.com/docs/atlas.en-us.apexref.meta/apexref/apex_methods_system_set.htm#apex_System_Set_methods.

Map

A classe `Map` não é uma coleção propriamente dita, mas faz parte do framework. Essa classe é especial por ter a capacidade de relacionar dois objetos, `Key` e `Value`. Em outras linguagens, como Python e C#, ela é conhecida como `Dictionary` (Dicionário). Veja a seguir as suas características:

- Não permitir chaves duplicadas;
- Não manter a ordem em que os elementos são adicionados;
- Permitir adicionar elementos, chave e valor, nulos.

Sintaxe

```
< Map<Key,Value> > <nomeVariavel> [ = new Map<Key,Value>() ];
```

Imagine que você precise identificar os nossos celulares pelo número de série, de forma que não possa haver mais de um `Celular` com o mesmo número de série. Veja a seguir como utilizar a classe `Map` para atender a essa necessidade:

```
Map<String, Celular> celularesPorNumeroDeSerie
                            = new Map<String, Celular>();

celularesPorNumeroDeSerie.put ('23432'
                        , new Celular (11973237237));

celularesPorNumeroDeSerie.put ('45545'
                        , new Celular (11983247336));

celularesPorNumeroDeSerie.put ('56565'
                        , new Celular (11993257135));

celularesPorNumeroDeSerie.put ('23432'
```

```
                                , new Celular (11973237238));

System.debug ( estados.size() ); // Exibe 3 elementos
```

Note que, ao definir o nosso `Map`, determinamos dois tipos: um corresponde à chave, `Key`, e o outro corresponde ao valor, `Value`. Dessa forma, fazemos a correlação entre duas instâncias de objetos, permitindo uma **indexação** ou **agrupamento de objetos**. Outra diferença do `Map` para outras coleções está na forma de adicionar elementos, pois no `Map` precisamos sempre correlacionar dois objetos através do método `put`.

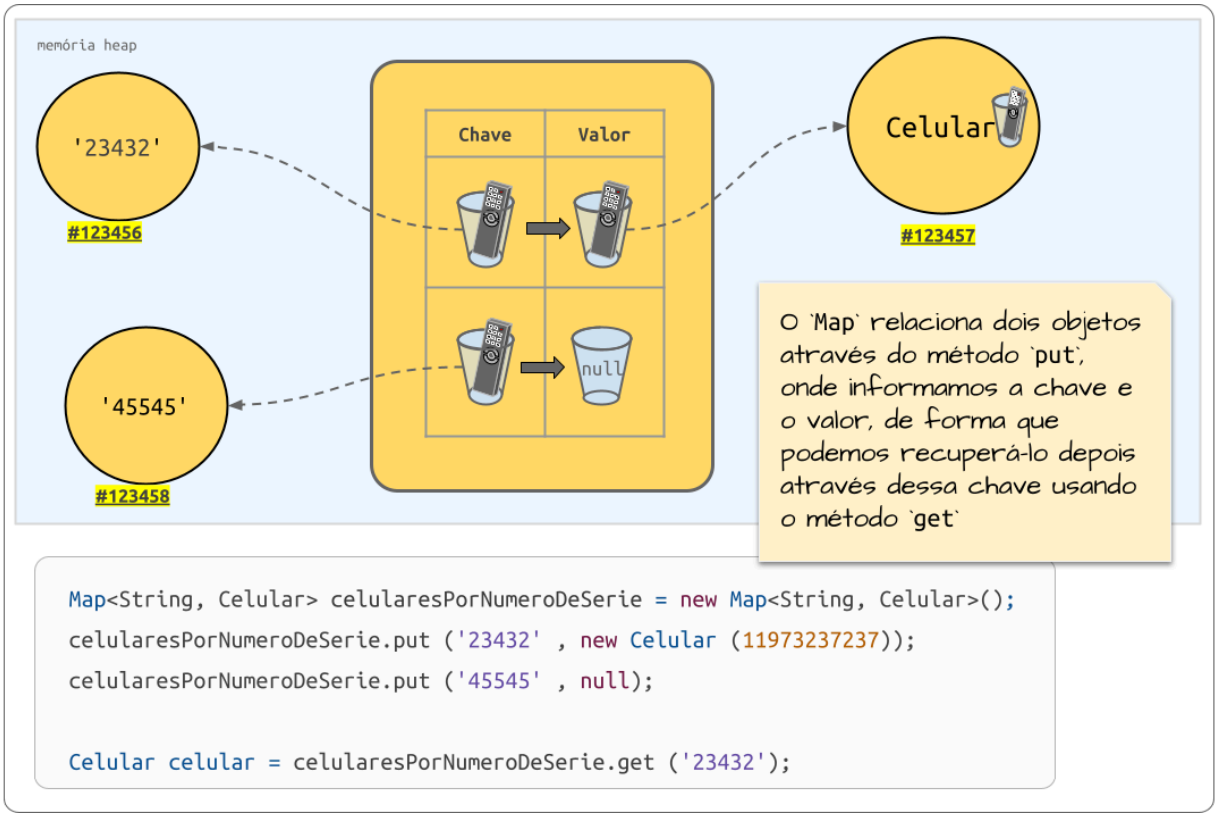


Figura 2.23: Visualizando o Map na Heap

Conheça sua API

Como comentado anteriormente, devemos dominar a linguagem que estamos utilizando. Por isso, devemos conhecer os métodos fornecidos pela classe `Map`. Veja a seguir alguns métodos:

- **`Object put(Object key, Object value)`** : adiciona uma nova composição de chave e valor. Caso já exista entrada com a chave informada, ele substitui o valor e retorna o valor determinado anteriormente;
- **`Boolean containsKey(Object key)`** : retorna `true` caso a chave passada já exista; caso contrário, retorna `false` ;
- **`Object get(Object key)`** : retorna o objeto correspondente à chave passada. Caso não exista a chave, é retornado `null` ;
- **`Set<Object> keySet()`** : retorna um `Set` com todas as chaves;
- **`List<Object> values()`** : retorna uma `List` com todos os valores.

#fica-a-dica 🤔

Geralmente utilizamos o `Map` com a intenção de **indexar** elementos ou **agrupá-los**.

Indexando com Map

Imagine que você tenha a necessidade de obter um cliente (`Customer`) pelo CNPJ. Podemos utilizar o `Map` para esse fim:

```
Map<String, Customer> customersByDocument
                            = new Map<String, Customer>();

customersByDocument.put ('18877615000166'
                , new Customer ('18877615000166'));

customersByDocument.put ('87423854000169'
                , new Customer ('87423854000169'));

customersByDocument.put ('18877615000166'
                , new Customer ('18877615000166'));

// a última linha identificou que já existia uma
// chave com o mesmo valor e trocou a referência
// do valor para a nova instância de Customer.
```

```
System.debug ( customersByDocument.size() ); // > 2

Customer customer = customersByDocument
                                .get('18877615000166');

System.debug ( customer.getDocument() ); // > 18877615000166
```

> **#se-liga-na-info** 😎
>
> Convencionamos o termo **indexação** quando temos somente uma chave relacionada a somente a um valor, dessa forma normalizamos a nomenclatura e convencionamos a nossa base de código.

Vimos um exemplo de indexação de dados, agora vamos ver como é um agrupamento de dados utilizando `Map`.

Agrupe dados com Map

Existem cidades que adotam um CEP único para todas as ruas/avenidas. Vamos utilizar a classe `Map` para agrupar todos os endereços (`Address`) por CEP (`zipCode`). Para isso, vamos trazer uma outra novidade, uma nova sintaxe do `for` específica para coleções:

```
Map<String, List<ShippingAddress>> addressesByZipCode
                    = new Map<String, List<ShippingAddress>>();

List<ShippingAddress> addresses = new List<ShippingAddress>();

addresses.add (
        new ShippingAddress('68908450'
                , 'Avenida Germano Silva Souza') );

addresses.add (
        new ShippingAddress('12220000'
              , 'Avenida Juscelino Kubitschek') );
```

```
addresses.add (
        new ShippingAddress('68908450', 'Rua São Paulo') );

addresses.add (
        new ShippingAddress('68908450'
            , 'Rua XV de Novembro 786') );


for (ShippingAddress address : addresses) {

    String zipCode = address.getZipCode();

    if ( !addressesByZipCode.containsKey ( zipCode ) ) {

        addressesByZipCode.put ( address.getZipCode(),
                                new List<ShippingAddress>() );

    }

    // a partir daqui todas as chaves têm ao menos uma lista.
    // Então podemos obter a lista de 'ShippingAddress'
    // com a mesma chave e adicionar um novo elemento.
    addressesByZipCode.get (address.getZipCode()).add (address);

}

// SHOW 4 - total de endereços
System.debug ( addresses.size() );

// SHOW 2 - total de chaves
System.debug ( addressesByZipCode.size() );

List<ShippingAddress> foundAddresses
                            = addressesByZipCode.get('68908450');

System.debug ( foundAddresses.size() ); // SHOW 3
```

Perceba que, quando falamos de agrupamento, temos uma estrutura semelhante a esta: `Map<Tipo, List<Tipo>>`. Ou seja, estamos mapeando uma chave para um conjunto de valores!

#fica-dica-clean-code

Gostaria de trazer uma convenção simples para deixar o nosso código mais limpo. Ela é aplicada ao controlador `if`, porém muitas vezes é ignorada por devs iniciantes.

Não utilize operador de comparação `==` com uma literal `true`.

Esse tipo de avaliação compara um valor `Boolean` com outro `Boolean`, que, no final, é o equivalente a `true == true`.

```
Boolean conectado = false;

if ( conectado == true ) {

    System.debug ('Condição verdadeira');

}
```

Essa avaliação é considerada como duplicada, pois podemos utilizar a própria variável como condição de avaliação, dado que o operador espera um valor `Boolean` ***true***. Observe a implementação a seguir:

```
// prefira
Boolean conectado = false;

if ( conectado ) {

    System.debug ('Condição verdadeira');

}
```

Outro **contraexemplo** que gosto de apresentar é quando comparamos uma variável `Boolean` com a literal `false`. Veja o exemplo a seguir:

```
Boolean conectado = false;

if ( conectado == false ) {
   System.debug ('Não está conectado');
}
```

Esse **contraexemplo** traz uma avaliação semelhante a `true == false`. Esse é um cenário onde devemos utilizar o operador de negação junto da variável a ser avaliada. Observe a implementação a seguir:

```
Boolean conectado = false;

//se não está conectado, então
if ( !conectado ) {
   System.debug ('Não está conectado');
}
```

Formas de inicializar as coleções

A linguagem APEX fornece alternativas facilitadas para inicializar as coleções. Tanto a classe `List` quanto a classe `Set` fornecem a mesma forma de inicialização; já o `Map` muda um pouco, pois temos a necessidade de informar a chave e o valor.

```
List<Celular> motorolas = new List<Celular> {
    new Celular (11978654321),
    new Celular (11987563412)
};

Set<Celular> iphones = new Set<Celular> {
    new Celular (11978654321),
```

```
    new Celular (11987563412)
};

Map<String, Celular> celularesPorNumero = Map<String, Celular> {
    '11978654321' => new Celular (11978654321),
    '11987563412' => new Celular (11987563412)
};
```

Logo após `new List <Tipo>`, em vez de utilizar os `()`, você utiliza as `{}` e adiciona os elementos separando-os por vírgula. Já para o `Map`, logo após a `{` informamos a `chave` seguida do operador `=>`, que indica qual é o `valor` e os novos valores também separados por vírgula (,).

#fica-dica-clean-code

Evite utilizar o tipo da coleção como parte do nome de sua variável, pois dessa forma você estará sempre atrelado àquele tipo específico. Utilize sempre o plural para representar uma coleção, seja uma `List`, `Set` ou `Map`. Veja o exemplo a seguir:

```
// ------ prefira -----------
List<Pedido> pedidos = new List<Pedido>();

Set<Id> ids = new Set<Id>();

Map<String, Pedido> pedidosPorEstado
                        = new Map<String, Pedido>();

// ------ em vez de -----------
List<Pedido> pedidoList = new List<Pedido>();
Set<Id> setId = new Set<Id>();
Set<Id> idSet = new Set<Id>();
Map<String, Pedido> mapPedidos
                      = new Map<String, Pedido>();
```

Conheça os tipos suportados pela linguagem APEX

A linguagem APEX possui diversos tipos, como *String*, *Integer*, *Double*, *Decimal*, *Boolean*, *Date*, *DateTime* e outros. É importante que você complete o seu estudo pela documentação da linguagem APEX, dado que este livro não será um manual dessa linguagem, mas focará em como usá-la da melhor forma.

Para saber mais informações sobre os tipos, acesse o Guia de Desenvolvimento APEX disponível em: https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/langCon_apex_datatypes_variables_intro.htm

2.8 Definindo visibilidade dos membros da classe

Quando vimos a primeira classe em APEX, deixei alguns pontos em aberto, sendo que um deles é o **modificador de acesso** `public`. Ele faz parte dos modificadores que determinam a visibilidade dos membros da classe, ou seja, são eles que determinam se uma classe ou membro dela podem ser vistos por outras classes. Veja a lista de modificadores a seguir:

- **`public`** : aplicável à classe e a seus membros, determina que outras classes podem acessar;
- **`private`** : aplicável à classe e a seus membros, determina que o membro não pode ser acessado por outra classe, somente pela própria classe;
- **`default`** : representa a ausência de modificador de acesso. Quando não for informado um modificador, o compilador assume que o membro é automaticamente privado, `private` ;
- **`protected`** : determina que o membro só pode ser acessado por uma classe filha, caso contrário será considerado `private`.

Veremos mais sobre esse modificador de acesso em breve.

Utilizando os modificadores

Nos exemplos apresentados anteriormente, definimos praticamente tudo como `public`. Isso porque eu gostaria que você testasse as classes através de execução anônima, mas vamos ver como podemos melhorar a visibilidade de nossos membros.

Apesar de a linguagem APEX possibilitar que uma classe seja `private`, em um cenário real basicamente temos as classes públicas. Dado que uma classe nasce para ser consumida, ou seja, utilizada por "alguém", então vamos convencionar que todas as classes serão `public`.

Quando abordamos modificadores de acesso, estamos tratando de encapsulamento, que podemos entender como uma forma de proteger o acesso a determinada informação ou operação.

> O termo *encapsulamento* é bem amplo e também remete à responsabilidade e "automagicamente" à coesão, mas por agora vamos aplicá-lo somente ao que se refere à proteção de acesso à informação.

Com a intenção de proteger os dados/operações em mente, vamos observar o cenário da classe `Pedido` a seguir, que fornece formas de calcular o valor total de um pedido:

```
public class Pedido {

    public Decimal total;

    public Decimal totalLiquido;

    public List<ItemPedido> items;
```

```
    // construtor omitido

    public void adicionarItem ( ItemPedido item ) {

        this.items.add(item);

        this.total += item.total;
        this.totalLiquido += item.totalLiquido;

    }

}
```

Todos os atributos da classe `Pedido` são públicos, o que permite que qualquer classe externa os acesse e os manipule. Veja o exemplo de consumo da classe `Pedido` a seguir:

```
Pedido pedido = new Pedido();

pedido.total = 1000.00;
pedido.totalLiquido = 980.00;
pedido.items = new List<ItemPedido>();

System.debug ('Total sem impostos: ' + pedido.totalLiquido );
System.debug ('Total Geral: ' + pedido.total );
```

Conseguimos burlar qualquer desejo de estabelecer uma regra de negócio para a classe `Pedido`, pois é possível manipular qualquer informação ou operação. O ideal é que somente a classe `Pedido` tenha acesso para modificar certos dados, pois **é dela a responsabilidade** de gerir tal informação.

Para isso, precisamos proteger os atributos da classe utilizando o modificador `private` e fornecer de forma controlada o acesso de somente leitura aos atributos `total` e `totalLiquido`:

```
public class Pedido {

    private Decimal total;
```

```
    private Decimal totalLiquido;

    private List<ItemPedido> items;

    // construtor omitido

    public void adicionarItem ( ItemPedido item ) {

        this.items.add(item);

        this.total += item.total;
        this.totalLiquido += item.totalLiquido;

    }

    public Decimal getTotal () {
        return this.total;
    }

    public Decimal getTotalLiquido () {
        return this.totalLiquido;
    }


}
```

Agora só podemos acessar os membros públicos da classe `Pedido`. Veja como ficou o consumo dela:

```
ItemPedido item = new ItemPedido();
item.quantidade = 1;
item.preco = 1000.00;
item.precoLiquido = 980.00;
item.total = 1000.00;
item.totalLiquido = 1000.00;

Pedido pedido = new Pedido();

pedido.adicionarItem ( item );
```

```
System.debug ('Total sem impostos:' + pedido.getTotalLiquido() );
System.debug ('Total Geral: ' + pedido.getTotal()  );
```

Conseguimos resolver o problema da classe `Pedido`, mas ainda temos os mesmos problemas em `ItemPedido`. Para corrigir a `ItemPedido`, vamos adotar a convenção de encapsulamento, conhecida como ***getter and setters***. Ela determina que os atributos da classe sejam privados e os métodos, públicos — `getXXX` e `setXXX`.

Veja a implementação de `ItemPedido` seguindo essa convenção:

```
public class ItemPedido {

    private Decimal quantidade;

    private Decimal preco;

    private Decimal precoLiquido;

    // construtor omitido

    public Decimal getQuantidade() {
        return this.quantidade;
    }

    public void setQuantidade( Decimal quantidade ) {
        this.quantidade = quantidade;
    }

    public Decimal getPreco() {
        return this.preco;
    }

    public void setPreco( Decimal preco ) {
        this.preco = preco;
    }
```

```
    public Decimal getPrecoLiquido() {
        return this.precoLiquido;
    }

    public void setPrecoLiquido( Decimal precoLiquido ) {
        this.precoLiquido = precoLiquido;
    }

    public Decimal getTotal() {
        return this.preco * this.quantidade;
    }

    public Decimal getTotalLiquido() {
        return this.precoLiquido * this.quantidade;
    }

}
```

A classe `ItemPedido` não fornece mais acesso direto a seus atributos. Se tentarmos executar o exemplo anterior, ele não compilará mais, indicando que não é possível acessar os atributos privados da classe.

#se-liga-na-info 😎

A convenção `getter` e `setter` é oriunda do **Java**, que adota essa forma de encapsulamento seguindo esta sintaxe:

```
public <Tipo> get<NomeDaVariavel> () {
    return this.<nomeDaVariavel>;
}

public void set<NomeDaVariavel> ( <Tipo> <nomeDaVariavel> ) {
    return this.<nomeDaVariavel> = <nomeDaVariavel>;
}
```

De uma forma geral, conseguimos proteger a informação utilizando a convenção *getter* e *setter* do Java com o modificador `private`. Mais à frente, abordarei outras formas de encapsulamento que ajudarão a proteger e garantir a coesão de nossas classes.

O que vem a seguir?

Agora que vimos como podemos aplicar **Orientação a Objetos** dentro da plataforma **Salesforce**, vamos ver um dos recursos mais poderosos da linguagem: como ela interage com o seu banco de dados.

Capítulo 3
Acessando dados através do APEX

3.1 Entenda o que são os Objetos da plataforma

Quando estamos desenvolvendo sistemas, precisamos armazenar os dados em algum lugar, seja em um banco relacional (como MySQL ou PostgreSQL) ou em bancos considerados NoSQL (como MongoDB). A plataforma Salesforce fornece o seu próprio banco de dados e, diferente dos bancos anteriormente citados, ela não armazena os dados em tabelas ou documentos, mas em **Objetos**, também conhecidos como `SObjects`.

Essa representação visa trazer uma linguagem mais simples e natural para o usuário final. De forma simples, basta chamar de **Objeto** aquilo que você conhece como *Tabela*.

A Salesforce traz como padrão diversos processos previamente mapeados, como os de pré-venda, venda, pós-venda, financeiro e muitos outros, isso tudo com a intenção de prover uma visão 360º para o negócio do cliente. Esses processos estão mapeados através de **Objetos padrões** (*Standard*) definidos pela plataforma. Outro ponto importante é a sua capacidade de extensão: podemos criar nossos próprios Objetos, chamados de **Objetos customizados** (*Custom Objects*).

Segue uma lista, **extremamente** reduzida, de alguns Objetos padrões (*Standard*):

- **Lead**: representa seu cliente em potencial, comumente chamado de *Prospect*;
- **Account**: principal objeto da plataforma, tudo gira em torno dele, que é a representação de seu cliente;

- **Contact**: representação do contato de um cliente;
- **Opportunity**: representa uma venda em potencial, algo que deve ser trabalhado junto com o cliente através de um funil de vendas, com a intenção de fechar ou não uma venda;
- **OpportunityLineItem**: determina os itens que estão sendo trabalhados na oportunidade (*Opportunity*).

Muitos sistemas são construídos, de forma generalizada, utilizando operações de SQL (*Select Query Language*), ou seja, realizando um `SELECT` no banco de dados relacional para obter os dados de uma determinada tabela. Na plataforma não é diferente, só que, em vez de utilizar SQL, vamos utilizar o **SOQL** (*Salesforce Object Query Language*).

3.2 Introdução a SOQL

A *Salesforce Object Query Language* é baseada no SQL ANSI (https://www.w3schools.com/sql/sql_intro.asp), mas com várias adaptações. Para ver a sua utilização, vamos começar com o cenário em que você necessita obter todas as *Contas* (`Account`) que são do estado de São Paulo, com o comando a seguir:

```
SELECT Id, Name, BillingStreet, BillingState, BillingCity
FROM Account
WHERE BillingState = 'São Paulo'
```

Bem simples, muito semelhante ao SQL. Mais à frente veremos mais detalhes e características do SOQL.

> Diferente de outras implementações de SQL, o SOQL não suporta o uso do caractere curinga `*` para obter os campos de um *objeto*. Mesmo em bancos SQL, esse recurso não é recomendado.

Vamos ver a sintaxe do SOQL de forma simplificada.

Sintaxe simplificada

```
SELECT fieldList [subquery][...]
FROM objectType
[WHERE conditionExpression]
[GROUP BY {fieldGroupByList}]
    [HAVING havingConditionExpression] ]
[ORDER BY fieldOrderByList {ASC|DESC} [NULLS {FIRST|LAST}]]
[LIMIT numberOfRowsToReturn]
[OFFSET numberOfRowsToSkip]
```

Para mais detalhes sobre a sintaxe do SOQL, consulte o *Developer Guide* disponível em: https://developer.salesforce.com/docs/atlas.en-us.soql_sosl.meta/soql_sosl/sforce_api_calls_soql_select.htm.

Padrão de nome de objetos e campos

Uma das características da plataforma Salesforce é o padrão de nomeação de objetos e campos customizados, que é adicionando um sufixo `__c` ao nome de API. Então, quando criamos um Objeto customizado chamado *City*, seu nome de API ficará `City__c` , e o mesmo critério se aplica a qualquer campo customizado criado.

3.3 Navegação e relacionamento entre objetos

Quando estamos desenvolvendo aplicações e temos a necessidade de obter dados de duas tabelas do banco de dados relacional, utilizamos instruções como `JOIN` em nossos *SELECTS*; porém, quando tratamos de Objetos Salesforce, a plataforma fornece um recurso muito poderoso, semelhante ao `JOIN` , que é a capacidade de navegação. Esse recurso permite obter os dados de objetos relacionados como *lookup*.

Por exemplo, o objeto *Contrato* (`Contract`) possui uma referência (*lookup*) ao `Account` . Então podemos obter os dados do `Account` de forma muito simples, utilizando uma navegação por *ponto* (.). Veja o exemplo a seguir:

```
SELECT Id, ContractNumber, AccountId , Account.Name,
Account.BillingAddress
FROM Contract
WHERE Account.Name = 'Edge Communications'
```

Perceba que podemos utilizar esse conceito de navegação de várias formas, seja para trazer conteúdo do objeto ou como critério de pesquisa.

#se-liga-na-info 😎

O termo *lookup* é semelhante a uma chave estrangeira (*foreign key*) de um banco relacional. Ele tem a capacidade de relacionar dois objetos/tabelas. Em consultas *SOQL*, podemos navegar somente até cinco (5) níveis.

Os campos do tipo *lookup* aplicados em Objetos padrões utilizam a composição do nome do objeto seguido da palavra `Id` . Já os campos customizados sempre serão sufixados com o `__c` . Mas quando utilizamos esses campos para navegação, temos a seguinte composição:

Nome do campo	Composição para navegação	Exemplo
`AccountId`	`Account`	`Account.Name`
`City__c`	`City__r`	`City__r.Name`
`Project__c`	`Project__r`	`Project__r.EndDate__c`

Nome do campo	Composição para navegação	Exemplo
`ContractId`	`Contract`	`Contract.ContractNumber`

Sempre que precisarmos navegar em campos customizados do tipo *lookup*, temos que substituir o `__c` por `__r`.

> A notação `__c` e `__r` traz uma certa verbosidade ao código, mas acredito que essa notação deva estar diretamente ligada à performance.

Observe o SOQL a seguir, que utiliza diversos *lookups* customizados no objeto `Account`.

```
SELECT Id
    , Name
    , BillingStreet
    , BillingCity__r.Name__c
    , CNAE__r.Description__c
    , CNAE__r.Code__c
FROM Account
WHERE Name = 'Edge Communications'
```

Vamos um pouco mais longe: suponha que o objeto *Cidade* (`City`) tenha um *lookup* para o objeto *Estado* (`State__c`), e que necessitamos obter em `Account` a cidade e o nome do estado, como no exemplo a seguir:

```
SELECT Id, Name, BillingStreet
    , BillingCity__r.Name__c
    , BillingCity__r.State__r.Name
    , CNAE__r.Description__c
    , CNAE__r.Code__c
FROM Account
WHERE Name = 'Edge Communications'
```

Note que o SOQL anterior navega até o nome do Estado, definido no objeto `State__c`. Nesse exemplo, temos três níveis de navegação.

Agora que você viu como é fácil obter dados de objetos relacionados (*lookups*) com uma relação de **muitos para um**, vamos ver a relação inversa, ou seja, como obter os contratos de cada conta do ponto de vista do objeto `Account`. Esse recurso é chamado de *inner select*.

```
SELECT Id, Name
    , ( SELECT Id
             , ContractNumber
             , StartDate
             , EndDate
        FROM Contracts )
FROM Account
WHERE Name = 'Edge Communications'
```

Essa *query* nos apresenta, mais uma vez, a diferença entre **Objetos padrões** (*Standard Objects*) e **Objetos customizados** (*Custom Objects*). Note que, para obter os contatos pertencentes a *Conta* (`Account`), utilizamos o nome do objeto no plural, `Contracts`. A convenção para *inner select* em Objetos padrões é sempre utilizar o nome do objeto no plural — por exemplo, `Contracts`, `Orders`, entre outros.

Para realizar um *inner select* em um Objeto customizado, devemos utilizar a notação `__r` junto do nome da API fornecido durante o mapeamento reverso. Veja a seguir a imagem que ilustra esse mapeamento durante a criação do campo *lookup*. Ela apresenta a criação do campo `Advisor` em `Account` e o momento em que determinamos as duas vias de comunicação entre os objetos.

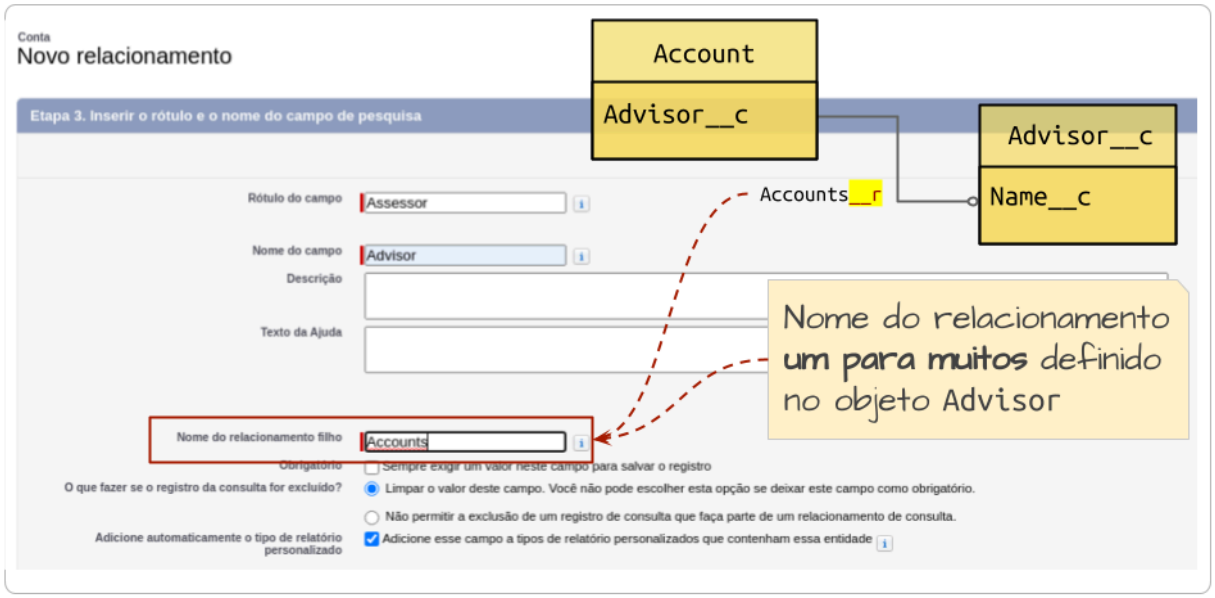


Figura 3.2: Nome do relacionamento reverso no objeto filho

Observe a *query* a seguir, na qual pesquisamos os *Assessores* (`Advisor`) juntos de suas *Contas* (`Account`):

```
SELECT Id
    , Name, Code__c
    , ( SELECT Id
            , Name
        FROM Accounts__r )
FROM Advisor__c
```

Agora que vimos como é simples a utilização de *inner selects* para obter mais de um objeto de forma simultânea, imagine um cenário onde precisamos obter todas as *Contas* (`Account`) que tiveram suas *Oportunidades* fechadas e ganhas. Para isso, podemos utilizar o recurso de uma *subquery* como critério de pesquisa. Observe na *query* a seguir onde utilizamos esse recurso:

```
SELECT Id, Name
FROM Account
WHERE Id IN ( SELECT AccountId
                FROM Opportunity
                WHERE StageName = 'Closed Won' )
```

Esse tipo de *subquery* só pode ser realizado em campos do tipo `Id`, isto é, em campos que sejam *lookup*. Utilizamos o operador `IN` ou `NOT IN` seguido da *subquery* declarada dentro dos parênteses `()`.

> **#se-liga-na-info** 😎
>
> Para realizar essas *queries* diretamente do VSCode, basta escrever a *query* em um novo arquivo, selecioná-la e executar `CTRL+Shift+P > SFDX: Execute SOQL Query with Currently Selected Text > REST API`.

3.4 Acessando os dados pelo APEX

A Salesforce focou em facilidade no consumo de dados, então podemos executar consultas em qualquer parte do código APEX. Esse ponto é positivo e negativo ao mesmo tempo, pois veremos que podemos fazer uma verdadeira bagunça no código com isso.

Vamos utilizar a mesma *query* mencionada anteriormente em uma parte do nosso código. Para isso, basta colocá-la dentro de colchetes `[]` e finalizar o comando com `;`, conforme o exemplo:

```
public List<Account> findByClosedWonOpportunities () {

    return [ SELECT Id, Name
            FROM Account
            WHERE
                Id IN ( SELECT AccountId
                        FROM Opportunity
                        WHERE
                        StageName = 'Closed Won')];

}
```

Esse código retorna uma `List` com o resultado da pesquisa. Caso não haja resultado, a `List` sempre será vazia e não nula.

Agora vamos ver como é que se dá o retorno de um único resultado. Para isso, usaremos a palavra reservada `LIMIT`, com a intenção de obter, no nosso exemplo, a primeira conta com o nome `Edge Communications`, ordenando o resultado (`ORDER BY`) pela data de criação (`CreatedDate`).

```
public Account findFirstEdgeCommunications () {

    return [ SELECT Id, Name
                FROM Account
                WHERE Name = 'Edge Communications'
                ORDER BY CreatedDate
                LIMIT 1];

}
```

Esse exemplo sempre retornará um único resultado; caso não encontre nenhum resultado, será lançada uma exceção `System.QueryException: List has no rows for assignment to SObject`.

> De forma bem resumida, **EXCEÇÃO** é um erro ocorrido durante a execução do código. No cenário apresentado, o APEX força o erro `System.QueryException`. Não se preocupe com isso por enquanto, pois veremos mais detalhes sobre exceções mais à frente.

Uma alternativa para obter um único registro é utilizar uma `List` como resultado da pesquisa. Veja no exemplo a seguir onde precisamos obter uma última *Oportunidade* (`Opportunity`) no estágio "Fechada e Perdida":

```
List<Opportunity> opportunities;

opportunities = [ SELECT Id
```

```
                    , LastModifiedDate
                FROM Opportunity
                WHERE StageName = 'Closed Lost'
                ORDER BY LastModifiedDate DESC
                LIMIT 1];

if ( !opportunities.isEmpty ()) {

    Opportunity opportunity = opportunities.get(0);

    Account account = new Account ( Id = opportunity.Id );

    account.LastClosedLostOpportunityDate__c =
opportunity.LastModifiedDate;

}
```

Podemos consumir o produto de uma *query* atribuindo diretamente a uma `List` e verificando se ela é nula ou não. No caso do exemplo anterior, se ela não for nula, atribuímos o valor da posição zero (`opportunities.get(0).LastModifiedDate`).

Agora veja o mesmo cenário implementado de outra forma, onde o resultado da consulta SOQL é atribuído diretamente a uma variável do tipo `Opportunity` . Note que aqui será utilizado um novo bloco de código, `try {} catch () {}` , que vamos conhecer mais à frente.

```
try {

    Opportunity opportunity;

    opportunity = [ SELECT Id
                        , LastModifiedDate
                    FROM Opportunity
                    WHERE StageName = 'Closed Lost'
                    ORDER BY LastModifiedDate DESC
                    LIMIT 1 ];

    Account account = new Account ( Id = opportunity.Id );
```

```
    account.LastClosedLostOpportunityDate__c
                = opportunity.LastModifiedDate;

} catch (QueryException e) {
    System.debug (e.getMessage ());
}
```

Nesse exemplo, quando fazemos a *query* que retorna um único resultado e atribuímos a uma variável que não seja do tipo de uma coleção, o APEX lança um erro (*Exception*) indicando que não foi possível obter o resultado.

#se-liga-na-info 😎

Nesta sua fase do aprendizado, aqui na primeira parte do livro, não se preocupe em escolher qual é a melhor forma, mas em entender quais formas existem para depois ver qual é a mais indicada para implementar códigos elegantes!

3.5 Aumente a segurança do seu código

Vários sistemas são invadidos por falta de segurança. Um exemplo simples de ataque, mas muito comum, é o *SQL Injection*, uma forma de atacar um banco de dados inserindo dados maliciosamente. E como evitar esses ataques dentro do seu código? Utilizando **variáveis de substituição** (*bind variable*). Em vez de utilizar valores fixos no código da SOQL, utilizamos variáveis.

Vamos ver o código com a utilização de *bind variables*:

```
String stage = 'Closed Lost' ;

List<Opportunity> opportunities;

opportunities = [ SELECT Id, LastModifiedDate
                    FROM Opportunity
                    WHERE
                        StageName = :stage
                    ORDER BY
                        LastModifiedDate DESC
                LIMIT 1];
// trechos omitidos...
```

Em códigos anteriores, estávamos passando o texto `Closed Lost` diretamente na consulta, de modo que ela era menos dinâmica e segura. Já o último código apresentado nos traz a mesma consulta utilizando a variável `stage` sendo atribuída, utilizando a sintaxe `:<nomeDaVariavel>`. Podemos utilizar também variáveis do tipo `List`, `Set`, `Date`, `DateTime`, entre outras, como forma de substituir os valores e tornar a *query* mais dinâmica e segura.

Segurança é uma preocupação constante para a plataforma Salesforce, que tem adicionado vários novos recursos para aumentá-la, como a cláusula `WITH SECURITY_ENFORCED`, adicionada na versão 48 da API. Esse recurso aplica ao resultado da *query* as regras de permissão de visualização de campos do usuário logado, lançando um erro caso não tenha acesso a determinado campo. Veja o mesmo código apresentado anteriormente, mas com a cláusula `WITH SECURITY_ENFORCED`:

```
String stage = 'Closed Lost' ;

List<Opportunity> opportunities;

opportunities = [ SELECT Id, LastModifiedDate
                FROM Opportunity
```

```
                WHERE
                    StageName = :stage
                WITH SECURITY_ENFORCED
                ORDER BY LastModifiedDate DESC
                LIMIT 1];
```

Um ponto de atenção sobre essa cláusula: ela deve ser utilizada com cautela, pois adiciona mais processamento na execução e pode causar lentidão. Então, devemos utilizar ou não essa instrução? Geralmente não a utilizo, pois quando um usuário não tem acesso a determinado campo da consulta, ele fica em branco, sem gerar erros. Mas isso não quer dizer que *você* não deva utilizá-la, só recomendo atenção ao contexto de sua aplicação.

3.6 Utilizando funções de agregação com SOQL

Imagine o cenário onde precisamos obter o total de vendas de um determinado período ou a média de consumo de um determinado produto. Nesses casos, utilizamos **funções de agregação**, e o SOQL tem diversas funções voltadas para isso, como: `SUM`, `COUNT`, `AVG`, `MIN`, `MAX`, entre outras.

Vamos demonstrar, através de uma consulta, como calcular o valor médio das oportunidades por cliente:

```
List<AggregateResult> averageOfSales;

averageOfSales = [ SELECT AccountId
                    , AVG(Amount) average
                FROM Opportunity
                GROUP BY AccountId ];

for ( AggregateResult result : averageOfSales ) {

    System.debug (  result.get('AccountId')
                    + '---'
```

```
                    + result.get('average') );

}
```

Quando utilizamos funções de agregação, o SOQL sempre retornará uma `List<AggregateResult>`, e não uma lista do objeto pesquisado. Com isso, a maneira de obter os dados da consulta também é alterada, pois o tipo `AggregateResult` tem a característica de um `Map`. Observe que, para obter o campo `AccountId`, é utilizado o método `.get(<NomeDoCampo>)`.

#fica-a-dica 🤔

Sempre que tiver a necessidade de realizar cálculos com base em dados, tente priorizar obtê-los calculados utilizando funções de agregação, e não o realizando diretamente no código. Dessa forma, poupamos memória e processamento.

3.7 Atualizando dados pelo APEX

Agora que vimos como é fácil consumir dados de Objetos do Salesforce, vamos ver como atualizá-los através do APEX. No exemplo apresentado, mencionamos o desejo de *atualizar* o dado de uma *Conta*, porém não realizamos essa operação propriamente dita. Vamos, então, completar o código com a operação de `update`:

```
List<Opportunity> opportunities;

opportunities = [ SELECT Id
                       , LastModifiedDate
                  FROM Opportunity
                  WHERE StageName = 'Closed Lost'
                  ORDER BY LastModifiedDate DESC
```

```
                LIMIT 1];

if ( !opportunities.isEmpty ()) {

    Opportunity opportunity = opportunities.get(0);

    Account account = new Account ( Id = opportunity.Id );

    account.LastClosedLostOpportunityDate__c =
opportunity.LastModifiedDate;

    update account;

}
```

Note que foi somente adicionado o comando `update`. Muito simples, não é? O APEX fornece os comandos `insert`, `update`, `delete`, `upsert` e `merge`. Esses comandos são os responsáveis pelas operações de atualização nos dados no banco, ou seja, operações de DML (*Data Manipulation Language*).

Os comandos `insert`, `update` e `delete` são autoexplicativos, mas os de `upsert` e `merge` podem ser mais complicados. Vamos conhecer melhor cada um deles.

O comando `merge`

O comando **`merge`** trata registros duplicados de `Lead`, `Account`, `Contact` ou `Case`. Podemos unificar até três registros em um só. Ele não é válido para outros objetos. Vamos ver um caso de uso.

Em um CRM, manter os dados de clientes o mais exatos possível é essencial para manter um bom relacionamento e, com isso, aumentar as chances de realizar uma nova venda. Podemos utilizar o comando `merge` para unificar registros duplicados em apenas um registro. Veja o exemplo a seguir, onde temos dois registros de conta praticamente iguais:

```
List<Account> accounts = new List<Account> {
    new Account (Name= 'Bisso Capital INC'),
    new Account (Name= 'Bisso Capital INC.'),
    new Account (Name= 'Bisso Capital')
};

insert accounts;

Account masterAccount = accounts.get(1);

List<Account> accountsToMerge = new List<Account> {
    accounts.get(0),
    accounts.get(2)
};

merge masterAccount accountsToMerge;
```

Nesse código, unificamos os registros duplicados em uma única conta, a *master*. Caso as outras contas possuíssem contatos, eles também seriam migrados para a conta *master*.

Para mais informações sobre esse comando, consulte o *Developer Hand Book*, disponível em: https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/langCon_apex_dml_examples_merge.htm.

O comando `upsert`

O comando `upsert` realiza uma atualização ou inserção com base no ID ou chave externa — não se preocupe com o termo "chave externa", que veremos logo à frente.

Vamos ver mais a fundo como o `upsert` funciona e como ele é versátil. Observe o código a seguir, onde utilizamos o controlador `if` para avaliar se devemos inserir ou atualizar um registro:

```
public Account save ( Account account ) {
```

```
    BrazilianDocumentValidator validator
                    = new BrazilianDocumentValidator ();

    if ( validator.isValid () ) {
        // valida e retorna um erro, omitido por simplicidade
    }


    if (account.Id == null) {
        insert account;
    } else {
        update account;
    }

    return account;
}
```

O exemplo desse código é um cenário muito comum quando tratamos de integração sistêmica, pois recebemos um dado de conta e precisamos aplicar uma determinada validação, aqui representada pela validação de CPF/CNPJ, e depois salvar o registro no banco.

Agora observe o mesmo exemplo utilizando o comando de `upsert` :

```
public Account save ( Account account ) {

    BrazilianDocumentValidator validator
                    = new BrazilianDocumentValidator ();

    if ( validator.isValid () ) {
        // valida e retorna um erro, omitido por simplicidade
    }

    upsert account; //SIMPLÃO !!!

    return account;
}
```

Veja que o código ficou mais simples, dada a versatilidade do `upsert`. Agora vamos analisar um cenário de negócio mais próximo do real.

A *Bisso Capital Inc.*, uma financeira, necessita manter os dados de *Clientes* sincronizados entre a Salesforce e seu sistema interno. Para isso, é necessário que, em ambos os sistemas, tenhamos uma referência (chave) forte para o cliente. É aí que entra o que chamamos de **chave externa**: um identificador, que pode ser único, que cria um índice para determinar os registros com base em uma chave de outro sistema. Veja na imagem a seguir como definimos um campo como chave externa:

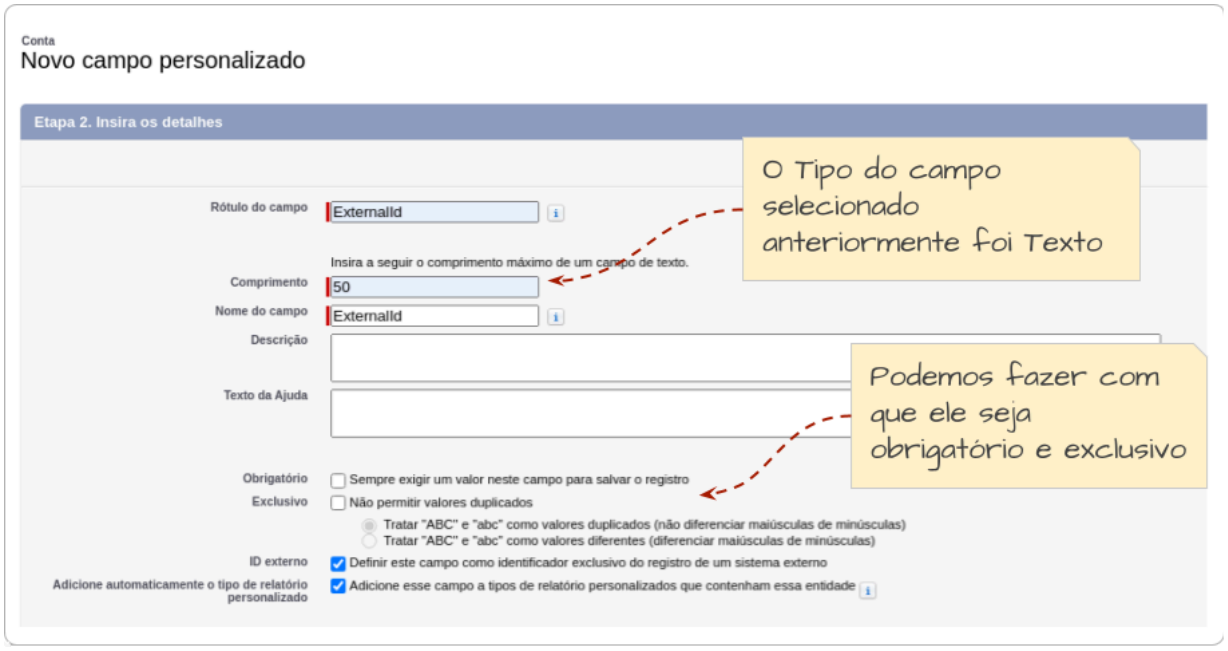


Figura 3.6: Campo de External Id em Conta

Agora que temos o contexto de negócio, vamos ver com um olhar um pouco mais técnico como faremos o sincronismo dos dados de *Conta*. Receberemos os dados de *Conta* com os dados de cidade de cobrança e o CNAE (*Código Nacional de Atividade Econômica*) da empresa. Esses campos estão definidos como *lookup* para `BillingCity__c` e `CNAE__c`, respectivamente.

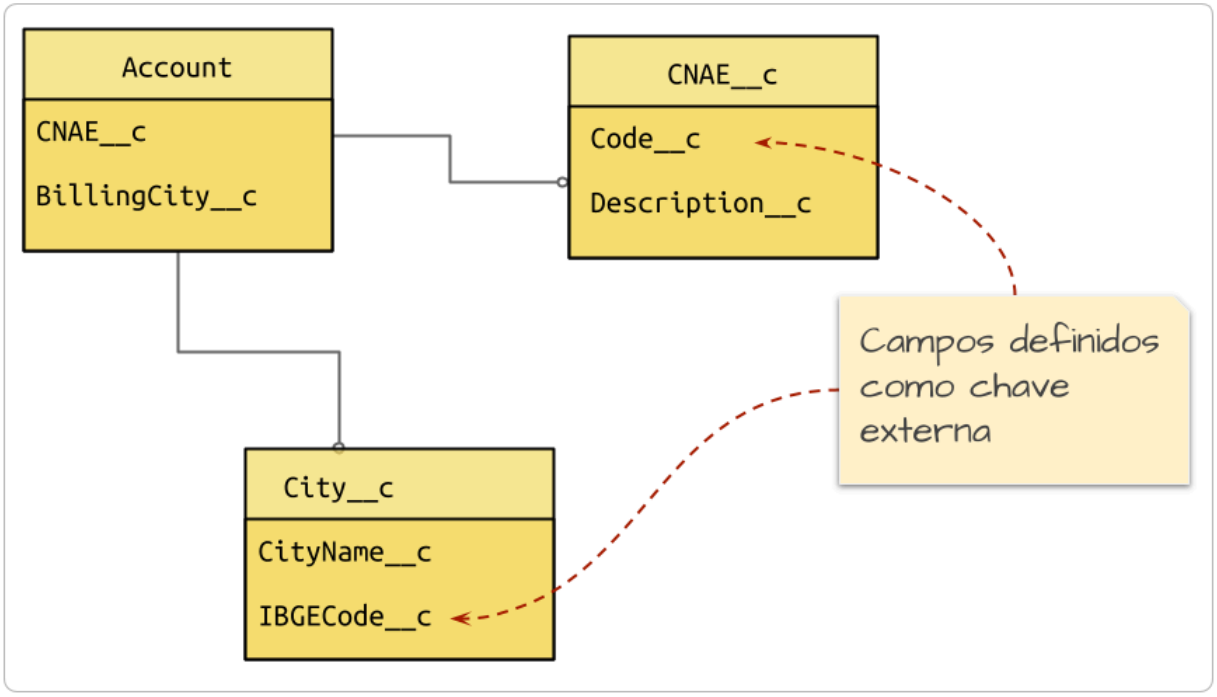


Figura 3.7: Diagrama de Relacionamento de Conta

Receberemos um dado semelhante a este, representado em JSON, para facilitar a visualização e o entendimento. Nele você pode notar a presença de uso do relacionamento (`__r`) para fazer a referência às chaves externas:

```
{
    "Name": "Super Mega Cliente1",
    "DocumentNumber__c": "62132916012",
    "ExternalId__c":  "62132916012",
    "CNAE__r": {
        "Code__c" : "29233923"
    },
    "BillingCity__r": {
        "IBGECode__c" : "50308"
    },
}
```

Se fôssemos desenvolver de forma tradicional, teríamos de fazer uma *query* para cada campo do tipo *lookup* em *Conta*, semelhante ao código a seguir:

```
public Account save ( Account account ) {

    // fluxo de validação omitido

    List<City__c> cities;

    cities = [ SELECT Id, IBGECode__c
            FROM City__c
            WHERE
                IBGECode__c = :account.BillingCity__r.IBGECode__c
            LIMIT 1];

    if ( !cities.isEmpty() ) {
        account.BillingCity__c = cities.get(0).Id;
    }

    List<CNAE__c> cnaes = [ SELECT Id, Code__c
                FROM CNAE__c
                WHERE
                Code__c = :account.CNAE__r.Code__c
                LIMIT 1];

    if ( !cnaes.isEmpty() ) {
        account.CNAE__c = cnaes.get(0).Id;
    }

    upsert account;

    return account;
}
```

O código não está complexo, mas aumentou um pouco. Agora pense se tivéssemos mais outros *lookups* - a complexidade aumentaria conforme o número de relacionamentos, o que tornaria o código difícil de manter e evoluir.

Veja a mesma operação, agora somente utilizando o `upsert` para informar qual chave externa ele vai utilizar para realizar a operação:

```
public Account save ( Account account ) {

    // fluxo de validação omitido

    upsert account ExtenalId__c; // NUUUOÓ!!! APELOU!!!

    return account;
}
```

Parece mágica, não é? O que acontece é que o comando de `upsert`, juntamente do nome do campo que representa a chave externa, faz um conjunto de operações:

- Ele realiza a pesquisa de cada *lookup* mapeando o *Id* automaticamente; caso não encontre o registro, ele retornará um erro;
- Ele determina se deve criar ou atualizar o registro do objeto informado levando em consideração o conteúdo do campo informado como chave externa; se encontrar um registro, ele o atualiza; senão, ele o cria. Caso encontre mais de um registro para a chave informada, ele retorna um erro.

Por conta de todas essas facilidades, utilizamos com frequência o `upsert` em fluxos de integração, já que cada sistema pode utilizar suas próprias chaves. Com isso, deixamos o mapeamento e o código muito mais simples.

Vimos como é simples realizar operações com o banco de dados na plataforma Salesforce, porém ainda não da forma ideal. Tenha calma, que ainda veremos como deixar nossos códigos mais elegantes. Siga para o próximo capítulo para ver como escutar essas atualizações no banco e compor soluções na plataforma.

Capítulo 4
Capture as atualizações do banco de dados

Nosso cliente *Bisso Capital Inc.* precisa garantir que as *Contas* (`Account`) tenham o número do CPF ou CNPJ válido. Não se trata de uma validação de tamanho ou formato: devemos fazer os cálculos de dígito verificador. Para isso, precisamos validar toda a criação ou atualização de uma *Conta* e, caso não atenda às regras de cada tipo de documento, o código retorna um erro.

Como podemos fazer tal validação? Simples: ouvindo as modificações ocorridas no objeto. Para isso, o APEX fornece um conceito muito similar ao de banco de dados, que é a utilização de **Triggers**. Elas escutam as modificações ocorridas no objeto e executam um bloco de código.

Vamos ver o exemplo:

```
/**
 * @author eduardo.bisso
 */
trigger Account on Account (before insert
                          , before update
                          , after insert
                          , after update) {

    System.debug('Account trigger update');

}
```

Entenda os eventos de atualização do banco

A ***Trigger*** nada mais é do que um bloco de código APEX que é executado antes e depois de cada evento de atualização. Ela tem uma sintaxe especial, onde você informa o nome da *Trigger* e, logo em seguida, o nome do Objeto Salesforce (`SObject`) seguido dos

eventos de atualização. Você pode utilizar os seguintes eventos, separados por vírgula:

- `before insert` : evento executado antes da inserção;
- `before update` : evento executado antes da atualização;
- `before delete` : evento executado antes da exclusão;
- `after insert` : evento executado depois da inserção;
- `after update` : evento executado depois da atualização;
- `after delete` : evento executado depois da exclusão;
- `after undelete` : evento executado depois de cancelar a exclusão.

Sintaxe da *Trigger*

```
trigger <TriggerName> on <Object> (
                            <trigger events> ) {

    // code block

}
```

Uma *Trigger* não é uma classe, mas, sim, um **metadado** com capacidade de executar um bloco de código. Inclusive, para criá-la, precisamos executar no *VSCode* o comando `SFDX: Create Apex Trigger` . Veja a seguir os passos para criar uma nova *Trigger*:

1. `CTRL+SHIFT+P` ;
2. `SFDX: Create Apex Trigger` ;
3. Informe o nome da *Trigger*;
4. Selecione `force-app/main/default` .

4.1 Utilizando as variáveis de contexto

Outra característica das *Triggers* são o que chamamos de **variáveis de contexto**. As variáveis de contexto só terão informação durante seu escopo de execução, permitindo o acesso a informações do ciclo de execução delas. Essas variáveis são acessadas pela classe

`Trigger`. Isso mesmo: temos uma classe APEX chamada `Trigger`, e é através dela que obtemos diversas informações sobre o estado de nossas *triggers*. Veja a seguir algumas dessas variáveis:

- **`Trigger.new`**: representa os registros que foram modificados, retornando uma `List` do tipo do objeto. Ex.: `List<Account> newAccounts = Trigger.new;` ;
- **`Trigger.oldMap`**: retorna um `Map<Id, SObject>` com a versão anterior dos registros atualizados.

Outras variáveis de contexto que indicam o estado da execução da *Trigger*, sempre retornando um `Boolean`, `true` ou `false`:

- **`Trigger.isExecuting`**: indica se a *Trigger* está em execução ou não;
- **`Trigger.isInsert`**: indica se está sendo executada uma operação de inserção;
- **`Trigger.isUpdate`**: indica se está sendo executada uma operação de atualização;
- **`Trigger.isDelete`**: indica se está sendo executada uma operação de exclusão;
- **`Trigger.isUnDelete`**: indica se está sendo executada uma operação de restauração de exclusão;
- **`Trigger.isAfter`**: indica se a *Trigger* está sendo executada antes de salvar o registro;
- **`Trigger.isBefore`**: indica se a *Trigger* está sendo executada depois de salvar o registro;
- **`Trigger.operationType`**: retorna uma *Enum* que unifica o indicador `isAfter` ou `isBefore` com a operação executada, conforme a lista a seguir:
  - `BEFORE_INSERT`
  - `BEFORE_UPDATE`
  - `BEFORE_DELETE`
  - `AFTER_INSERT`
  - `AFTER_UPDATE`
  - `AFTER_DELETE`
  - `AFTER_UNDELETE`

4.2 Utilizando a Trigger para fazer validações complexas

Agora vamos juntar tudo o que vimos sobre as *Triggers* e aplicar ao cenário proposto inicialmente, para garantir que não tenhamos CPF/CNPJ inválido no banco de dados.

Para isso, vamos utilizar a classe `BrazilianDocumentValidator`, que fará toda a validação na nossa nova *Trigger* em *Conta*, `Account`. Essa validação vai provocar um erro na tela ao tentar informar um *Documento* inválido:

```
/**
 * @author eduardo.bisso
 */
trigger Account on Account (before insert
                          , before update) {

    // determina quais registros foram atualizados.
    // coleção sempre no plural.
    List<Account> newAccounts = Trigger.new;

    BrazilianDocumentValidator validator
                           = new BrazilianDocumentValidator();

    //Identifica qual o evento/operação foi executado.
    switch on Trigger.operationType {

        when BEFORE_INSERT, BEFORE_UPDATE {

            // itera sobre as contas para aplicar as validações
            for ( Account account : newAccounts ) {

                if ( !validator
                        .isValid(account.DocumentNumber__c) ) {

                    // adiciona um erro caso não seja válido
                    account
```

```
                        .DocumentNumber__c
                        .addError ('Documento Inválido' );

                }

            }

        }

    }

}
```

O código anterior ficou um pouco mais complexo do que os anteriores, então vamos vê-lo por partes. Primeiramente, para identificar os eventos de `before insert` e `before update`, foi utilizado o controlador de fluxo `switch`, avaliando a *enum* `Trigger.operationType`. Dessa forma, determinamos a ação e o evento em uma única operação.

Após a determinação da ação/evento, podemos aplicar a validação propriamente dita para cada item contido na lista `newAccounts`, através do método `validate` da classe `BrazilianDocumentValidator`. Essa implementação está disponível no repositório: https://github.com/ercarval/salesforce-book-snippets/blob/feature/cap-3/force-app/main/default/classes/BrazilianDocumentValidator.cls.

Por fim, quando o CPF/CNPJ não é válido, será adicionado um erro ao campo, através do `account.DocumentNumber__c.addError ()`. Veja a imagem a seguir com o erro sendo apresentado na tela de criação/edição de *Conta*.

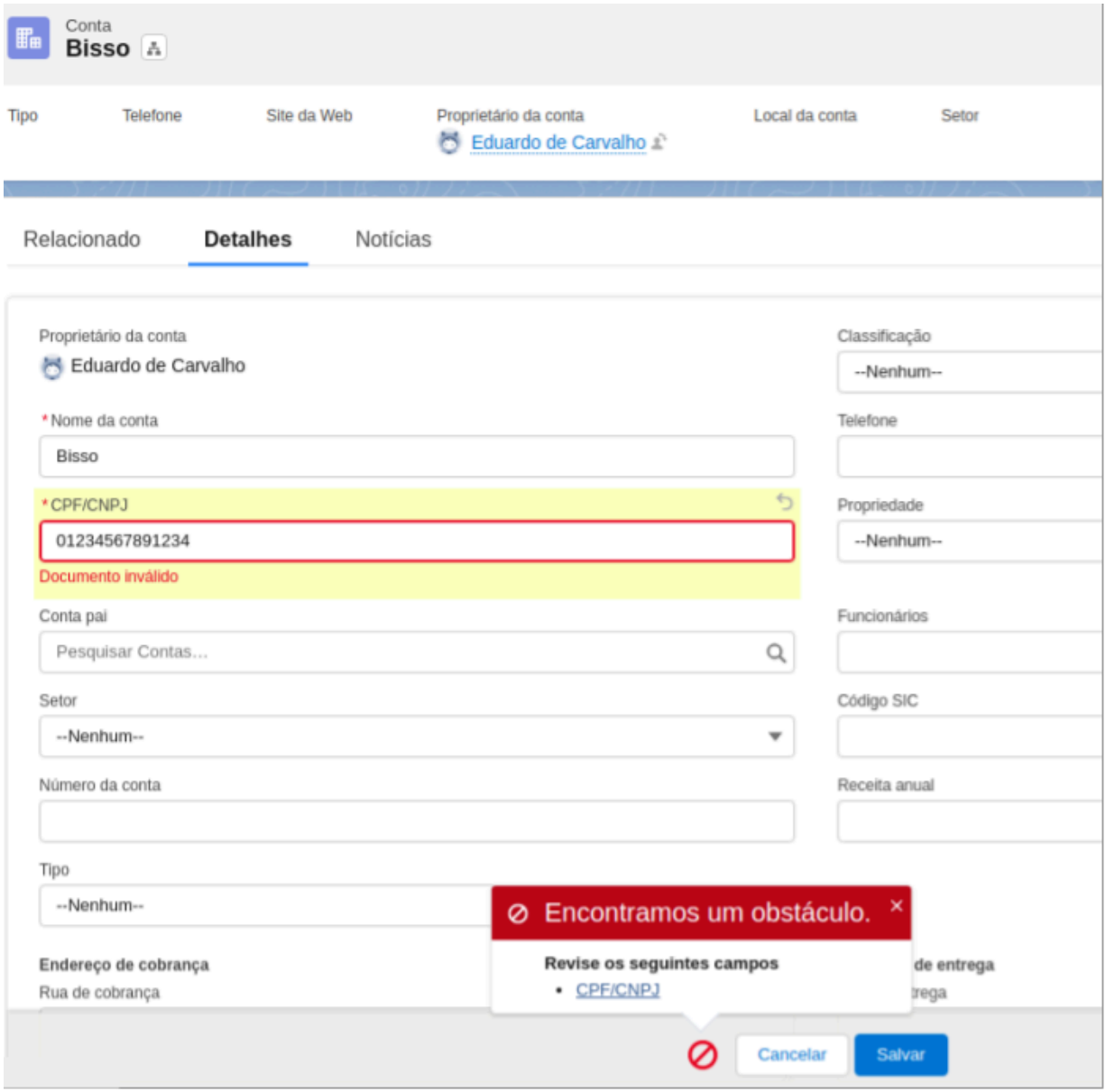


Figura 4.1: Erro ao informar um Documento inválido

Nesse fluxo de validação, utilizamos o `account.DocumentNumber__c.addError()` para indicar que havia um erro no campo. Poderíamos ter utilizado o método `addError()` também com `account.addError()`, indicando que o registro todo tem um problema.

Esses comportamentos `addError()` são um dos superpoderes que os Objetos Salesforce possuem. Isso porque todo `Account`, `Case`,

`Opportunity` e outros são, na verdade, o que chamamos de `SObject`, uma classe especial da Salesforce que representa seus objetos. Mais à frente veremos que ela nos fornece um conjunto de recursos como o `addError`.

4.3 Enriqueça dados com Triggers

A *Bisso Capital Inc.* tem a política de entrar em contato com todo novo Cliente, e essa atividade é realizada pelo Assessor da Conta. Com esse contato, é possível determinar o perfil do Cliente e definir a melhor estratégia de investimentos para ele.

Para apoiar essa operação, nos foi pedido que uma *Tarefa* (`Task`) fosse criada para o *Assessor* a toda nova *Conta*. Dessa forma, podemos rastrear o contato realizado pelo time e também apoiá-lo para um melhor atendimento ao cliente.

Agora que temos nosso cenário de negócio, vamos atender a essa necessidade através da *Trigger* criada anteriormente em *Conta*, ouvindo o evento de criação — no caso, será utilizado o `AFTER_INSERT`. Para cada ocorrência de uma nova *Conta*, criaremos uma nova *Tarefa* (`Task`) preenchendo os campos abaixo:

- Assunto (`Subject`): "Lembre-se de entrar em contato com o Cliente";
- Descrição (`Description`): "Entrar em contato para agendar a primeira reunião";
- Data e Hora de Início (`StartDateTime`): agendar para 1 dia após a data de criação do Cliente;
- Relativo a (`WhatId`): `Id` do Cliente em questão.

```
/**
 * @author eduardo.bisso
```

```
 */
trigger Account on Account (before insert
                          , before update
                          , after insert ) {

    List<Account> newAccounts = Trigger.new;

    BrazilianDocumentValidator validator;

    switch on Trigger.operationType {

        // código de validação omitido

        when AFTER_INSERT {

            List<Task> tasks = new List<Task>();

            for ( Account account : newAccounts ) {

                Task task = new Task();

                task.Subject = 'Lembre-se de entrar em'
                            + 'contato com o Cliente '
                            + account.Name;

                task.Description = 'Entrar em contato para'
                                 +'agendar a primeira reunião';

                task.ActivityDate = Date.today().addDays(1);
                task.WhatId = account.Id;
                tasks.add (task);

            }

            insert tasks;

        }

    }
```

```
}
```

Se comparada ao código de validação anterior, essa nova implementação ficou bem mais complexa. Se não tivéssemos omitido no exemplo a parte de validação, seria bem mais difícil de entender o comportamento da *Trigger*. Nesse cenário, utilizamos o `AFTER_INSERT` para capturar o `Id` gerado em `Account`, dado que isso só é possível após salvar o registro, no `after`. Recomendo que utilize sempre o `after` para esse fim, ou seja, sempre que for preciso criar/atualizar um outro objeto, utilize o `AFTER_INSERT` / `AFTER_UPDATE`.

Quando utilizamos as *Triggers* para criar/atualizar novos objetos, podemos dizer que estamos *enriquecendo* esses objetos. Esse comportamento de enriquecimento também pode ser dado quando temos de adicionar/atualizar campos no próprio objeto. Caso fôssemos enriquecer o próprio objeto, poderíamos utilizar os eventos `BEFORE_INSERT` ou `BEFORE_UPDATE`, dado que o registro ainda não foi atualizado. Isso pouparia processamento e um novo ciclo de atualização.

Nos cenários apresentados, vimos que as **Triggers** podem ser aplicadas para validações complexas, enriquecimento do mesmo objeto ou de outro. Vimos também como o código pode crescer com poucas regras de negócio. Nesta etapa, quero que você aprenda *como fazer*, e não *a melhor forma* — mas não se preocupe, veremos como sair de códigos como esse aplicando técnicas e padrões para deixar nosso código "*bonito e cheiroso*".

Siga em frente para o próximo capítulo, onde veremos como testar tudo o que fizemos até então.

Capítulo 5
Teste tudo!

Como você garante que tudo o que foi desenvolvido está funcionando?

Essa é uma questão muito complexa, mas, na verdade, não garantimos. De acordo com um post realizado pela *Smart Bear* (https://smartbear.com/blog/software-bug-cost/), erros em software custaram à economia global uma "bagatela" de quase $1.1 trilhão de dólares em 2016. É muito dinheiro!

Justamente porque negligenciamos essa questão, temos várias desculpas para dar: "Não dá tempo!", "Custa muito caro!", entre outras. Mas o fato é que uma hora a conta chega e o problema aparece, e vem no formato de bugs no software. E o que acontece? Aparece **tempo**! E dinheiro, só que são contados como negativos.

Pensando nisso, a Salesforce criou um mecanismo que impede que qualquer código APEX seja implantado em produção sem ter uma cobertura mínima de **75% de testes**.

Mas o que é cobertura de testes? Em nosso contexto, *cobertura* quer dizer que teremos classes de teste que vão executar as classes de negócio, com a intenção de garantir o seu funcionamento, seja para garantir o sucesso da execução como também a falha. Isso é o que chamamos de **testes automatizados**.

Quando falamos de testes automatizados, precisamos ter em mente que o teste deve ser executado da mesma maneira e produzir o mesmo resultado de forma constante. Ele também deve ser rápido, pois não queremos esperar uma hora para ver se algo funcionou.

Não tem como deixar de mencionar a prática de TDD (*Test-Driven Development*), prática criada por Kent Beck e Ron Jeffries a partir de suas experiências no desenvolvimento. Ela é focada em manter o

feedback contínuo do estado do software, ao se utilizar de testes para prover informações para tomada de decisões de forma simples, rápida, eficaz e incremental, maximizando a qualidade e o design do software.

De forma simples, essa prática tenta trazer uma maior empatia durante o ciclo do desenvolvimento, pois faz com que você pense inicialmente em como um método ou classe será consumido. Isso faz com que o código seja mais fácil de ser entendido, utilizado e, consequentemente, testado.

Agora que você já entende a importância de realizar testes em software e o princípio da empatia, da prática do TDD, vamos ver como implementá-los em APEX para garantir que nossos códigos estão funcionando de fato.

5.1 Testes de unidade

É difícil de pensar em unidade de software inicialmente, porque temos a tendência de ver somente aquilo com que temos algum contato. Uma tela, por exemplo, é somente a ponta do iceberg. Ela é a somatória de todas as partes, classes, arquivos *.js*, *.css* etc. Pense em sua aplicação como um conjunto de blocos de montar. Ela é composta por dezenas de peças que, juntas, compõem o todo. Veja a imagem a seguir:

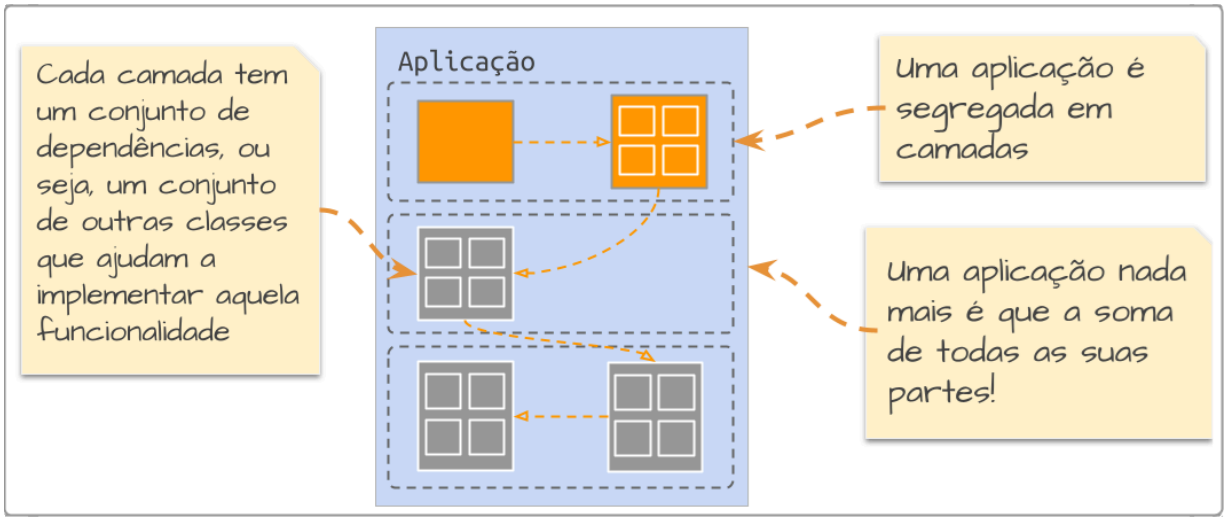


Figura 5.1: Unidades de uma aplicação

Os blocos são nossas *classes*, que, quando unidas, formam a aplicação. Com essa ideia de blocos em mente, vamos reavaliar o cenário apresentado da *Trigger* em `Account`, onde precisamos validar o CPF ou CNPJ. Nesse exemplo, nossa *Trigger* dependia da classe `BrazilianDocumentValidator` para realizar a validação.

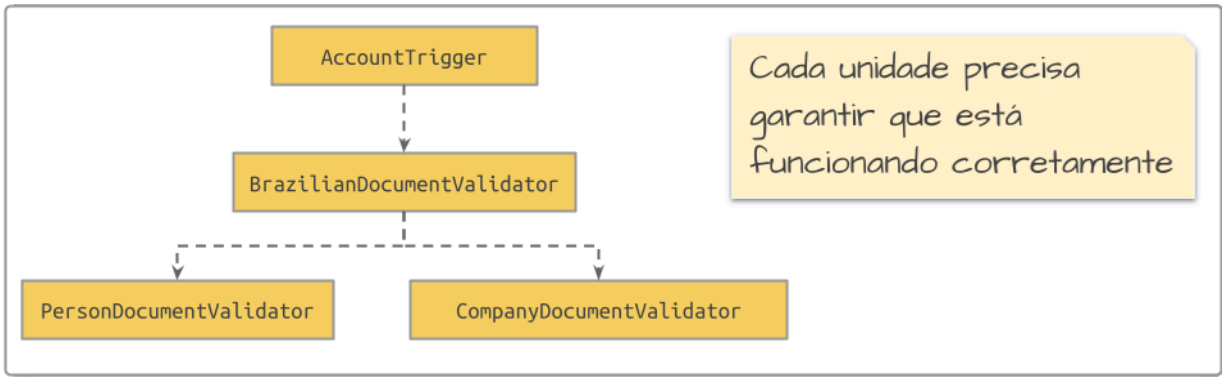


Figura 5.2: Hierarquia de classes da Trigger de Account

No caso apresentado, cada classe é uma unidade inicialmente, então cada classe deve garantir o seu funcionamento através de testes. O termo "unidade" também pode ser aplicado a um método, como uma microunidade de código.

O que testar?

Quando temos vários códigos previamente implementados, há uma maior dificuldade em identificar o que precisa ser testado. Recomendo que olhe primeiramente as dependências de forma inversa, conforme a imagem a seguir:

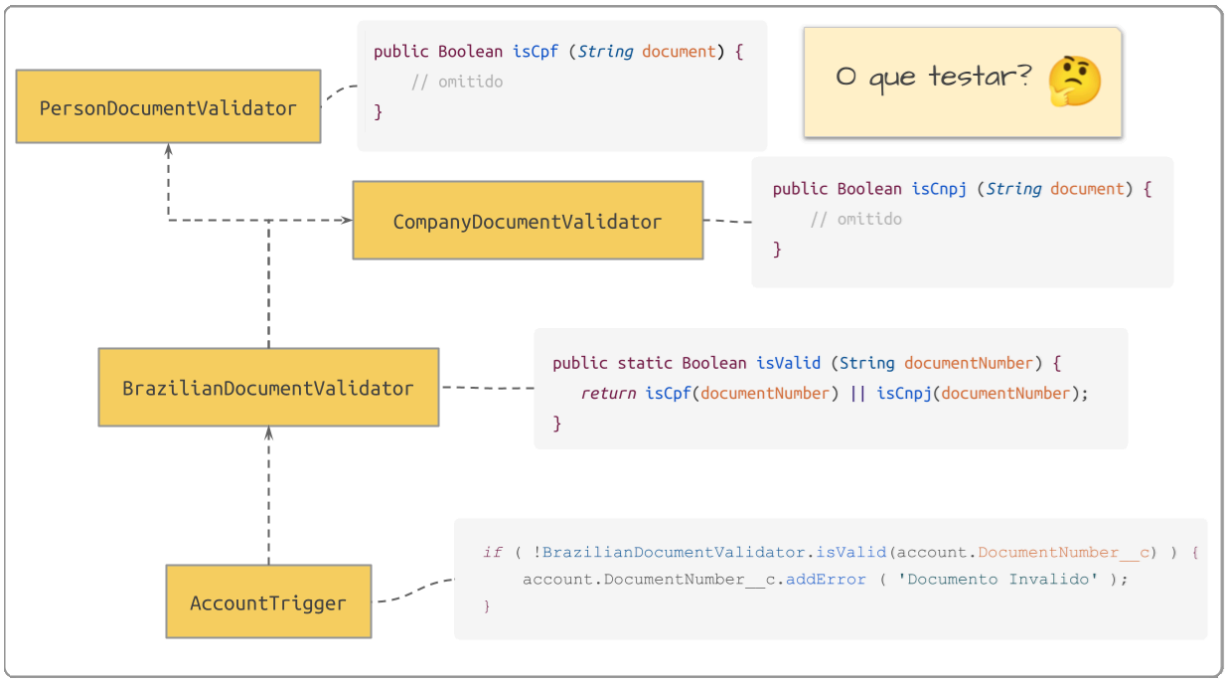


Figura 5.3: Árvore de dependência invertida

Quando ilustramos, fica mais clara a relação entre as classes. Nesse sentido, devemos começar pelo último ponto de dependência e seguir cobrindo todas as classes. Dessa forma, temos uma pequena lista de funcionalidades que precisam ser testadas:

1. O método `isCpf()` da classe `PersonDocumentValidator`;
2. O método `isCnpj()` da classe `CompanyDocumentValidator`;
3. O método `isValid()` da classe `BrazilianDocumentValidator`;
4. A criação de *Contas*.

Apesar de termos uma lista de classes que necessitam ser testadas, vamos olhar mais a fundo a `PersonDocumentValidator`. Ela necessita validar o CPF e possui um conjunto de regras:

- Aceitar somente números;

- Ter no máximo 11 dígitos;
- Não permitir CPFs com todos os dígitos iguais, como '111.111.111-11';
- Validação dos dois últimos dígitos, conforme as regras do Ministério da Fazenda.

Veja a seguir a implementação da classe `PersonDocumentValidator`:

```
/**
 * @author eduardo.bisso
 */
public class PersonDocumentValidator {

    private Integer MAX_SIZE = 11;

    public Boolean isCPF ( String cpf ) {

        if (String.isEmpty(cpf)) return false;

        cpf = removeMaskChars (cpf);

        if ( isNotNumeric (cpf)
                || isGreaterThanMaxSize (cpf)
                || isSameDigits (cpf)) {

            return false;

        }

        cpf = paddingLeftWithZero(cpf);

        List<String> digits = cpf.split('');

        Integer digit10 = calculateDigit( digits, 10, 9 );
        Integer digit11 = calculateDigit( digits, 11, 10 );

        return ( digit10 == toInt( digits, 9 ) )
                 && ( digit11 == toInt( digits, 10 ) );
    }
```

```
String removeMaskChars (String value) {
    return value.replaceAll ( '\\.|-', '' );
}

Boolean isNotNumeric ( String value ) {
    return !value.isNumeric();
}

Boolean isGreaterThanMaxSize ( String value ) {
    return value.length() > MAX_SIZE;
}

Boolean isSameDigits ( String value ) {
    return Pattern.matches('^(\\d)\\1*$', value);
}

String paddingLeftWithZero (String value) {
    return value.leftPad(11, '0');
}


Integer toInt (List<String> values
                                , Integer position ) {
    return Integer.valueOf( values.get(position) );
}

Integer calculateDigit( List<String> cpf
                            , Integer weight
                            , Integer iteration ) {

    Integer sum = 0;
    Integer rest;
    Integer num;

    for ( Integer i = 0; i < iteration; i++ ) {

        num = toInt ( cpf , i );
        sum += ( num * weight );
```

```
            weight--;

        }

        rest = 11 - ( Math.mod( sum, 11 ) );

        if ( rest == 10 || rest == 11 ) {
            return 0;
        }

        return rest;

    }

}
```

Agora que temos a implementação completa da classe `PersonDocumentValidator` em mãos, vamos ver como fazemos para testá-la.

Como testar?

Sempre que você for realizar um teste de uma classe/método, pergunte-se: qual o objetivo daquela implementação? O que ela recebe como parâmetro e qual a sua saída? Com essas informações, você pode iniciar a construção de sua classe/método de teste.

Para facilitar a resposta para a pergunta do propósito da classe, vou repetir a lista de requisitos que devemos cumprir para validar o CPF:

- Aceitar somente números;
- Ter no máximo 11 dígitos;
- Não permitir CPFs com todos os dígitos iguais, como '111.111.111-11';
- Validação dos dois últimos dígitos, conforme as regras do Ministério da Fazenda.

Juntamente às regras, vamos rever alguns trechos da implementação do método `isCPF`.

```
    public Boolean isCPF ( String cpf ) {

        if (String.isEmpty(cpf)) return false;

        // outras partes omitidas
    }
```

Com isso em mente, podemos criar uma nova classe, `PersonDocumentValidatorTest`. Ela utilizará a anotação `@isTest` para marcá-la como um teste.

> Toda classe que está anotada com `@isTest` não é contabilizada como um código produtivo, ou seja, não faz parte dos 75% de cobertura de código exigido pela Salesforce.

Iniciaremos justamente pelo teste que garante não validar um CPF nulo ou vazio:

```
/**
 * @author eduardo.bisso
 */
@isTest
public class PersonDocumentValidatorTest {

    @isTest
    public static void shouldReturnFalseForNullOrEmptyCpf () {

        PersonDocumentValidator validator
                        = new PersonDocumentValidator();

        System.assert (!validator.isCpf(null)
                    , 'CPF Nulo NÃO é válido' );

        Assert.isFalse ( validator.isCpf(null)
                    , 'CPF Nulo NÃO é válido');

        System.assert ( !validator.isCpf('')
```

```
                    , 'CPF Vazio Não é válido' );

        Assert.isFalse ( validator.isCpf('')
                    , 'CPF Vazio NÃO é válido');


    }
}
```

Note que o método `shouldReturnFalseForNullOrEmptyCpf` também está anotado com `@isTest`, indicando para a classe que ele deverá ser executado durante o ciclo de testes. Os métodos anotados com `@isTest` deverão ser `static void`. Como você pode ver no código anterior, basicamente executamos o método `isCpf()` e passamos um valor que deveria ser considerado como inválido.

Outro ponto importante é que **DEVEMOS SEMPRE** avaliar o resultado por meio de *asserções*, para identificar se o resultado obtido é de fato o desejado, utilizando os seguintes métodos da classe `System`:

- `assert( Boolean )`: afere se o valor passado é `true`; caso seja `false`, ele gera um erro de teste. Esse método também tem uma variação onde você pode passar uma mensagem em caso de erro;
- `assertEquals( expected, actual )`: compara se os dois primeiros argumentos são iguais; caso não sejam, gera um erro de teste. Esse método também tem uma variação onde você pode passar uma mensagem em caso de erro:
- `assertNotEquals( expected, actual )`: é o contrário do método `assertEquals`, pois avalia se os dois primeiros argumentos são diferentes; caso contrário, gera um erro de teste. Esse método também tem uma variação onde você pode passar uma mensagem em caso de erro.

A versão 56 da API adicionou uma nova classe, `Assert`, para fazer asserções. Ela traz novos métodos mais expressivos do que os

métodos da classe `System`, como o método `areEqual`, `isFalse`, entre outros.

Em nossa implementação, utilizei ambas as classes, `System` e `Assert`, justamente para demonstrar a pequena diferença entre elas. A classe `Assert` diminui a complexidade cognitiva, trazendo de forma direta a intenção da avaliação executada.

```
System.assert (!validator.isCpf(null)
            , 'CPF Nulo NÃO é válido' );


Assert.isFalse ( validator.isCpf(null)
            , 'CPF Nulo NÃO é válido');
```

Voltando para nossa implementação dos testes, podemos seguir a próxima avaliação que é a remoção dos caracteres " . " e " - ". Para isso, foi criado um método privado, `removeMaskChars`, que utiliza uma expressão regular para remover esses caracteres.

Esse é um cenário interessante, onde podemos evoluir a prática do teste de unidade. Podemos realizar cenários específicos para esses métodos garantindo que eles estejam funcionando de forma adequada. Para realizar essa avaliação, devemos modificar o método `removeMaskChars` adicionando uma nova anotação `@TestVisible`. Ela indica que o método privado é visível em classes de Teste (`@isTest`).

```
public Boolean isCPF ( String cpf ) {

    if (String.isEmpty(cpf)) return false;

    cpf = removeMaskChars (cpf);

}

@TestVisible
String removeMaskChars (String value) {
```

```
    return value.replaceAll ( '\\.|-', '' );
}
```

Com o método `removeMaskChars` visível podemos adicionar mais um método de teste em nossa classe recém-criada.

```
// Deve remover caracteres de máscara (###.###.###-##)

@isTest
public static void shouldRemoveMaskChars () {

    PersonDocumentValidator validator = new PersonDocumentValidator();

    String result = validator.removeMaskChars('XXX.XXX-AAA');

    Assert.areEqual ( 'XXXXXXAAA' , result  );

    // Caractere "/" inválido não será removido
    result = validator.removeMaskChars('111/ZZZ-XX');

    Assert.areEqual ( '111/ZZZXX' , result );

}
```

Essa implementação avalia o cenário onde são removidos os caracteres "." e "-", além da **NÃO** remoção de outros caracteres. Dessa forma conseguimos validar a execução do método `removeMaskChars` de forma isolada, garantindo todas as variabilidades.

Podemos seguir em frente em nossa validação do método `isCPF`, onde vamos cobrir as seguintes regras de validações:

- Aceitar somente números;
- Ter no máximo 11 dígitos;
- Não permitir CPFs com todos os dígitos iguais, como '111.111.111-11';

```
public Boolean isCPF ( String cpf ) {
```

```
    //testado
    if (String.isEmpty(cpf)) return false;

    //testado
    cpf = removeMaskChars (cpf);

    if ( isNotNumeric (cpf)
            || isGreaterThanMaxSize (cpf)
            || isSameDigits (cpf)) {

        return false;

    }

}
```

Esse é um cenário interessante de teste, pois temos três operações para serem avaliadas. Podemos aplicar a técnica adotada anteriormente, porém, como os métodos têm pouca variabilidade e necessitamos garantir que o CPF tenha essas regras aplicadas, vamos testar somente o método `isCPF` aplicando essas regras.

Começaremos pela validação que invalida documentos com caracteres não numéricos:

```
@isTest
public static void shouldReturnFalseForCpfWithText () {

    PersonDocumentValidator validator
                = new PersonDocumentValidator();

    String cpf = '111.444.777-XX';

    Boolean result = validator.isCpf(cpf);

    Assert.isFalse( result
            , 'CPF 111.444.777-XX NÃO é válido' );

    cpf = '111.444/777-11';
```

```
    Assert.isFalse( validator.isCpf( cpf )
            , 'CPF 111.444/777-11 NÃO é válido' );

}
```

Note que nesse teste utilizamos também uma outra formatação que fará com que o documento também seja inválido.

Agora vamos cobrir a validação do número de dígitos contidos em um CPF:

```
@isTest
public static void shouldReturnInvalidLengthForCpf () {

    PersonDocumentValidator validator
                = new PersonDocumentValidator();

    String cpf = '111.444.777-111';

    Boolean result = validator.isCpf(cpf);

    Assert.isFalse( result
            , 'CPF 111.444.777-111 NÃO é válido' );

    cpf = '9111.444.777-11';

    Assert.isFalse( validator.isCpf( cpf )
            , 'CPF 9111.444.777-11 NÃO é válido' );

}
```

Chegamos agora à última condição, onde garantimos que não haverá documentos com números repetidos. Vamos criar um modelo de texto com 11 `#`, que serão substituídos por um valor numérico de 0 a 9; assim, garantimos que não haverá sequência numérica repetida:

```
@isTest
```

```
public static void shouldRejectSameDigits () {

    PersonDocumentValidator validator
                    = new PersonDocumentValidator();

    String cpf = '###.###.###-##';

    for (Integer i = 0; i < 10; i++) {

        Boolean result
                = validator.isCpf(
                        cpf.replaceAll('#', i + ''));

        Assert.isFalse ( result
                       , 'CPF é Inválido' );

    }

}
```

Enfim chegamos ao último e o mais complexo ponto de validação: a verificação dos dois últimos dígitos. Primeiro, vamos repassar a regra de cálculo matemático para validação desses dígitos.

Validando o primeiro dígito

Primeiramente, deve-se multiplicar os 9 primeiros dígitos pela sequência decrescente de números de 10 a 2 e somar os resultados. Após a soma dos resultados, devemos obter o resto da divisão da soma por 11. Em seguida, subtrair o resto da divisão por 11.

Veja o exemplo do cálculo com o CPF `111.444.777-35` :

```
1*10 + 1*9 + 1*8 + 4*7 + 4*6 + 4*5 + 7*4 + 7*3 + 7*2 = 162

O resto da 162 / 11 é 8.

Cálculo final do dígito: 11 - 8 = 3
```

Nesse exemplo, o resultado da última subtração foi o número 3, que de fato corresponde ao primeiro dígito logo após o caractere "-", conforme o CPF `111.444.777-3X` . Se o resultado da última subtração for 10 ou 11, devemos retornar o dígito `0` .

Validando o segundo dígito

O cálculo do segundo dígito é muito semelhante ao primeiro, variando na sequência de dígitos. Devemos multiplicar os 10 primeiros pela sequência decrescente de números de 11 a 2 e somar os resultados. O resto do algoritmo é o mesmo, obtendo o resto da divisão.

Veja o exemplo do cálculo com o CPF `111.444.777-35` :

```
1*11 + 1*10 + 1*9 + 4*8 + 4*7 + 4*6 + 7*5 + 7*4 + 7*3 + 3*2 = 204

O resto de 204 / 11 é 6.

Cálculo final do dígito: 11 - 6 = 5
```

Com toda essa explicação em mente, veja novamente a implementação do método `calculateDigit` :

```
Integer calculateDigit( List<String> cpf
                        , Integer weight
                        , Integer iteration ) {

    Integer sum = 0;
    Integer rest;
    Integer num;

    for ( Integer i = 0; i < iteration; i++ ) {

        num = toInt ( cpf , i );
        sum += ( num * weight );
        weight--;

    }
```

```
    rest = 11 - ( Math.mod( sum, 11 ) );

    if ( rest == 10 || rest == 11 ) {
        return 0;
    }

    return rest;

}
```

Para avaliar se ele está correto, podemos anotá-lo com `@TestVisible` e realizar um teste de unidade para garantir o seu funcionamento.

```
@isTest
static void shouldCalculateFirstCpfDigit () {

    PersonDocumentValidator validator
                    = new PersonDocumentValidator();

    String cpf = '11144477735';

    List<String> digits = cpf.split('');

    Integer digit = validator.calculateDigit( digits, 10, 9 );

    Assert.areEqual ( 3 , digit );

}

@isTest
static void shouldCalculateSecondCpfDigit () {

    PersonDocumentValidator validator
                    = new PersonDocumentValidator();

    String cpf = '11144477735';

    List<String> digits = cpf.split('');

    Integer digit = validator.calculateDigit( digits, 11, 10 );
```

```
    Assert.areEqual ( 5 , digit );

}


@isTest
static void shouldCalculateCpfDigitWithZeroRest () {

    PersonDocumentValidator validator
                    = new PersonDocumentValidator();

    String cpf = '86989551003';

    List<String> digits = cpf.split('');

    Integer digit = validator.calculateDigit( digits, 10, 9 );

    Assert.areEqual ( 0 , digit );

}
```

Agora podemos "fechar a conta e passar a régua" com o método de teste do ciclo completo validando um CPF:

```
@isTest
public static void shouldAcceptValidCpfNumber () {

    PersonDocumentValidator validator
                = new PersonDocumentValidator();

    String cpf = '331.099.440-65';

    Boolean result = validator.isCpf(cpf);

    Assert.isTrue( result
            , 'CPF 331.099.440-65 é um CPF válido' );

    cpf = '221.935.320-60';
```

```
    result = validator.isCpf(cpf);

    Assert.isTrue( result
            , 'CPF 221.935.320-60 é um CPF válido' );


}
```

Agora vamos executar todos os testes da classe, conforme a imagem a seguir:

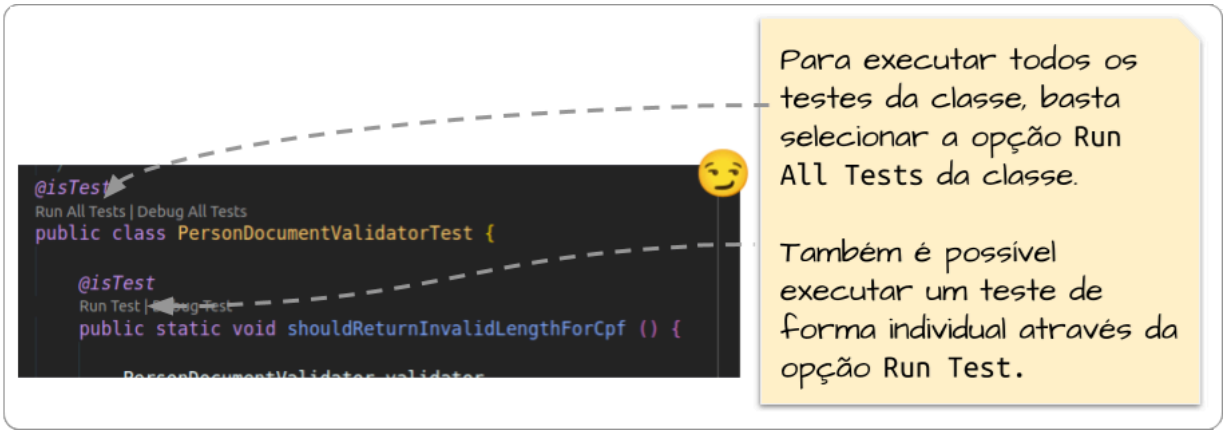


Figura 5.4: Executando os testes de uma classe

Obtemos um resultado lindo! 100% de cobertura com a nossa classe de testes.

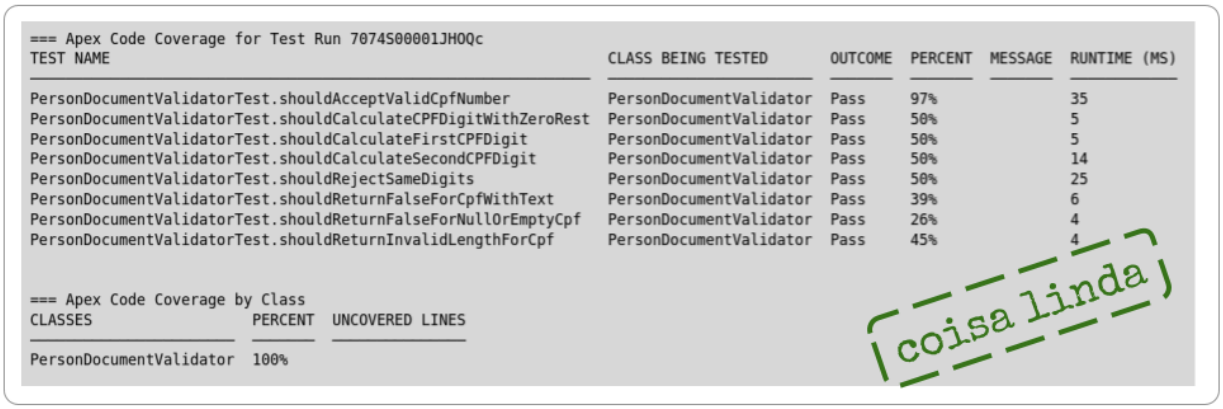


Figura 5.5: Teste com 100% de cobertura

5.2 Testando operações com o banco de dados

Agora vamos avançar em nossos testes com cenários que envolvem dados do banco. Para isso, vamos testar a nossa *Trigger*. Precisamos fazer uma análise semelhante à que fizemos ao testar a classe `PersonDocumentValidator`, identificando a entrada e a saída.

Para que uma *Trigger* possa ser acionada, é necessário que seja realizada uma operação de DML, que, no caso da *Trigger* de `Account`, é uma operação de inserção de dados. Já o resultado obtido dessa inserção, conforme a nossa implementação, é a criação da *Conta* juntamente de uma *Tarefa* (`Task`).

Então vamos criar uma nova classe de testes, a `AccountTriggerTest`, e vamos inicialmente criar um novo registro e verificar se ele foi criado com sucesso.

```
/**
 * @author eduardo.bisso
 */
@isTest
public class AccountTriggerTest {

    @isTest
    public static void shouldCreateAccountWithValidDocumentAndCreateTask
() {

        //dado (given)
        Account expectedAccount =
                new Account(Name = 'Teste',
                    DocumentNumber__c = '111.444.777-35');

        // quando (when)
        insert expectedAccount;

        // então (then)
        Assert.isNotNull (  expectedAccount.Id );
```

```
        // verificar se a Tarefa foi criada

        List<Task> tasks = [ SELECT Id
                    , Subject
                    , Status
                    , WhoId
                    , WhatId
                    FROM Task
                    WHERE WhatId = :expectedAccount.Id];

        Assert.isFalse ( tasks.isEmpty() ); // retornou resultado

        System.assertEquals ( 1
                , tasks.size()
                , 'Apenas uma tarefa deverá ser criada' );

    }

}
```

Agora começou a ficar mais sério, porque você precisa pensar no fluxo completo da *Trigger*. No caso, como estamos fazendo uma criação, ela vai executar os eventos de `BEFORE_INSERT` e `AFTER_INSERT`, um seguido do outro, por isso fizemos mais de uma verificação.

1. `System.assert ( expectedAccount.Id != null );`
2. `System.assert ( !tasks.isEmpty() );`

Você pode estar se perguntando: o que acontece com esses registros que são criados durante a execução do teste? Eles são "apagados" no final de cada método `@isTest`, ou seja, é realizada uma operação de *rollback* dos dados a cada execução.

Mas você pode também se perguntar: por que não utilizar os dados que já estão na Org para realizar os testes? Você até pode, mas não deve, pelo simples motivo de que os testes têm de **produzir o mesmo resultado de forma constante**. Isto é, não podemos garantir que os dados estejam perfeitos para consumo e garantir que estejam no **estado** esperado.

A palavra-chave é **CONSISTÊNCIA**: o mesmo resultado deve ser obtido sempre.

Vamos ver agora como fica o cenário do teste de falha no escopo de *Trigger*. Para avaliar esse cenário, será necessário utilizar um bloco de `try {} catch () {}`. Basicamente, o bloco `try` determina que o código definido dentro dele pode gerar um erro. Esse erro pode ser capturado por blocos específicos utilizando o comando `catch`, informando o tipo da `Exception`.

Para gerar esse cenário, vamos inserir um CPF inválido e verificar se ele não foi criado. Lembre-se de que, nesse caso, não devemos criar o registro, queremos que falhe.

```
@isTest
public static void
shouldGenerateErrorWhenTryToCreateAnAccountWithInvalidDocument () {

    //dado (given)
    Account expectedAccount = new Account(Name = 'Teste',
                        DocumentNumber__c = '111.444.777-95');

    try { // <--- tenta executar o bloco de código

        // quando (when)
        insert expectedAccount; // aqui deve gerar um erro

        Assert.fail('Não deveria criar a conta com CPF inválido');

    } catch (DmlException e) { //se for gerado esse tipo de erro
                               // executa o bloco de código

        // verifica se o erro é de Documento Inválido.
        Assert.isTrue (
                e.getDmlMessage(0)
                    .contains('Documento Inválido') );
```

```
    }


}
```

Esse código deverá provocar uma falha durante a inserção do registro de *Conta*, por isso foi adicionada uma **asserção**, `Assert.fail()`, logo após o comando de `insert`, indicando que nosso teste falhou.

Ao inserir um CPF inválido, deverá ocorrer um erro do tipo `DmlException`. Esse erro será capturado pelo bloco `catch (DmlException e)`. É por esse motivo que incluímos uma asserção `Assert.isTrue ()` dentro do bloco `catch`, indicando que o cenário do teste é obter uma falha desse tipo.

Por fim, para garantir o cenário, avaliamos se o erro gerado estava condizente com o que era esperado através do comando `e.getDmlMessage(0).contains('Documento Inválido')`.

E agora, acabou? Não! Precisamos validar ainda o fluxo de atualização, afinal, lembre-se de que a nossa *Trigger* também trata o evento de `BEFORE_UPDATE`.

Então vamos agora criar um novo cenário, que insere uma *Conta* com sucesso e depois, se o campo `DocumentNumber__c` for atualizado com um número válido, o registro deve ser atualizado com sucesso.

```
@isTest
public static void shouldUpdateAccountWithValidDocument () {

    //dado (given)
    Account expectedAccount = new Account(Name = 'Teste',
                        DocumentNumber__c = '111.444.777-35');

    insert expectedAccount;

    expectedAccount.DocumentNumber__c = '023.544.756-08';
```

```
    // quando (when)
    update expectedAccount;

    // então (then)
    System.assertEquals ( '023.544.756-08'
                , expectedAccount.DocumentNumber__c );
}
```

Esse cenário ficou muito semelhante ao primeiro, com a diferença de que o nosso fluxo de preparação do dado ficou mais extenso, pois precisamos inserir, fazer uma modificação no campo `DocumentNumber__c` , para depois realizar a ação de atualizar o registro.

Estamos quase acabando! Para fechar, vamos seguir para um fluxo de falha ao atualizar uma *Conta*. Ele será bem semelhante ao teste de inserção de CPF inválido, porém agora, ao realizar um `update` em uma *Conta* com um CPF inválido, deverá ser gerado um erro na atualização.

```
@isTest
public static void
shouldGenerateErrorWhenTryToUpdateAnAccountWithInvalidDocument () {

    //dado (given)
    Account expectedAccount = new Account(Name = 'Teste',
                        DocumentNumber__c = '111.444.777-35');

    insert expectedAccount; // aqui deve gerar um erro

    expectedAccount.DocumentNumber__c = '111.444.777-95';


    try { // <--- tenta executar o bloco de código

        // quando (when)
        update expectedAccount;

        Assert.fail (
            'Não deveria atualizar a conta com CPF inválido');
```

```
    } catch (DmlException e) { //se for gerado esse tipo de erro
                               // executa o bloco de código

        // vamos verificar se o erro de fato é de Documento Invalido.
        Assert.isTrue ( e.getDmlMessage(0)
                              .contains('Documento Inválido') );

    }

}
```

Agora que cobrimos todos os fluxos da *Trigger*, você pode utilizar o código do https://github.com/ercarval/salesforce-book-snippets/tree/feature/cap-5-11 para fazer a cobertura das outras classes, `BrazilianDocumentValidator` e `CompanyDocumentValidator`.

> **#se-liga-na-info** 😎
>
> O percentual de 75% de cobertura de testes é referente a todas as linhas de código, cobertas por testes da Org, não considerando classes anotadas `@isTest`, comentários e linhas em branco. Chamamos as classes que não são de teste de **classes produtivas**.

Com isso, fechamos a primeira parte do livro, onde tivemos bastante código e uma visão de fluxo maior sobre diversas etapas do desenvolvimento dentro da plataforma seguindo à risca convenções de código Java. No entanto, os códigos implementados até aqui ficaram verbosos e não tão elegantes. Isso porque focamos nos *fundamentos*, e não na melhor forma de implementá-los.

Gosto de dizer que essa primeira etapa foi ao estilo "Sr. Miyagi", ou, para os mais novos, "põe o casaco, tira o casaco". Já a próxima etapa traz uma visão mais profunda, onde focamos em melhores

práticas e aplicação de padrões de projeto para justamente combater esse código verborrágico e pouco elegante.

Parte 2: Bem-vindo(a) ao lado Orientado a Objetos da Força

> "*Para cada erro com o qual você aprende, você economizará milhares de erros semelhantes no futuro. Portanto, fique animado com esses erros e trate-os como oportunidades de aprendizado que produzem melhorias rápidas. Mas se você os tratar como coisas ruins, vai fazer a si mesmo e aos outros miseráveis, e não irá crescer.*"
>
> — Ray Dalio, autor do *best seller Princípios*

Agora que você já entendeu o que é a linguagem APEX e vários de seus recursos, como `SOQL`, `DML` e `Triggers`, bem como a testar seu código, é hora de deixá-los mais elegantes, *"bonitos e cheirosos"*!

Para isso, vamos nos aprofundar em conceitos extremamente importantes da Orientação a Objetos, como sobrecarga de parâmetros, herança, classes abstratas, interfaces, polimorfismo, entre outros. Aqui começa o seu processo de transformação: nosso código que era feio passará a aplicar padrões de projeto e ter uma característica produtiva.

Os padrões de projetos nada mais são que soluções recorrentes para problemas em um determinado contexto. Então, se você tem um problema e uma forma recorrente para solucioná-lo, você tem um padrão de projeto, que pode ser replicado.

É basicamente o que veremos nesta etapa: como melhorar nossa base de código, aplicando práticas de refatoração e padrões de projeto, com muita Orientação a Objetos.

Capítulo 6
Definindo papéis e responsabilidades

Lembra dos códigos anteriores, de *Triggers*, consultas SOQL? Estava tudo misturado! Não sabíamos de fato qual era o papel e qual a responsabilidade de cada classe, ou seja, o que cada classe deveria fazer. Esse conceito de responsabilidade em código também pode ser chamado de **coesão**, mas é difícil definir o que é coesão, então primeiro vamos ver o que *não é*.

Classes não coesas são aquelas que fazem de tudo um pouco, mas nada bem. Geralmente, classes ou métodos com pouca coesão tendem a ser grandes, com muitos blocos de código aninhados e repletos de distrações no formato de código. Esse tipo de situação nos distancia do propósito original do método ou classe.

Mas o que são classes coesas? São aquelas que contêm uma única responsabilidade.

Acho que "*bugueI*" sua mente! Esse é um conceito quase que filosófico. Então, para deixar menos abstrato, veja o cenário de negócio proposto e sua implementação em seguida.

CENÁRIO DE ADITIVO CONTRATUAL

Nossa cliente, a *Bisso Capital Inc.*, possui *Contratos* (`Contract`) referentes a serviços prestados pela financeira. Esses *Contratos* não poderão mais ser alterados após a sua assinatura. Para alterá-los, é necessário um aditivo, que é basicamente o *Contrato* original, com novas cláusulas. Esse aditivo também é representado pelo objeto `Contract` , porém com o tipo de registro *Aditivo*.

O *Contrato* do tipo *Aditivo* passará pelo mesmo fluxo e só será válido após a coleta de todas as assinaturas, passando a ser a única versão ativa desse *Contrato*. Sistemicamente, isso quer dizer que o *Contrato* do tipo *Aditivo* está "Ativo" e o anterior deixará de ser válido, ficando "Inativo".

Com esse cenário em mente, o código a seguir vai ativar os novos *Contratos* e inativar os antigos. Este trecho de código roda dentro de uma *Trigger* no `AFTER_UPDATE` , então as variáveis `newContracts` e `oldContracts` representam os novos valores de *Contratos* e os valores anteriores, respectivamente:

```
List<Contract> amendmentContracts = new List<Contract> ();

// Determina quais são os contratos
// que foram alterados para Assinado
for ( Contract contract : newContracts ) {

    Contract oldContract = oldContracts.get ( contract.Id );

    // avalia se o status do contrato mudou para Assinado
    // e possui um contrato anterior
    if ( contract.Status == 'Assinado'
        && oldContract.Status != 'Assinado'
        && contract.OriginalContract__c != null ) {

        amendmentContracts.add ( contract );
```

```
    }

}


if ( !amendmentContracts.isEmpty() ) {

    // obtém os contratos originais com base nos ids
    List<String> originalContractIds = new List<String> ();

    for (Contract contract : amendmentContracts) {

        originalContractIds.add (
                    contract.OriginalContract__c );

    }


    // consulta os contratos originais para inativação
    List<Contract> originalContracts =
                    [ SELECT Id, Status
                        FROM Contract
                        WHERE Id IN: originalContractIds ];

    // determina a relação do contrato original
    // com o contrato assinado

    if ( !originalContracts.isEmpty() ) {

        Map<Id, Contract> indexedOriginalContracts =
                      new Map<Id, Contract>(originalContracts);

        // determina o contrato original com base
        // no novo contrato assinado e
        // atualiza o contrato original como Inativado
        for (Contract amendmentContract : amendmentContracts ) {

            Contract originalContract =
```

```
                indexedOriginalContracts.get (
                        amendmentContract.OriginalContract__c);

            originalContract.Status = 'Inativado';

        }

        update originalContracts;

    }

}
```

Esse código ficou grande demais e, infelizmente, a maioria das Orgs Salesforce sofre de códigos infinitamente piores. É o seu papel reverter esse cenário apresentando códigos elegantes, aplicando o conceito de **coesão**.

Vamos listar tudo o que o código anterior está fazendo:

- Determina os aditivos atualizados que estão com o status `Assinado`;
- Determina os `Ids` dos contratos originais;
- Realiza a consulta dos contratos originais;
- Atualiza os contratos originais como inativados.

Bastante coisa! Agora, será que tudo o que essa parte do código está fazendo é responsabilidade dela? Vamos por partes, como diria o "Jack". :) Primeiro, vamos olhar para as consultas e para as atualizações em banco. Será que esse código precisa ser o responsável por construir e realizar a consulta? Será que outras classes não precisariam de uma consulta semelhante?

A resposta para primeira pergunta é **NÃO**: a classe não deveria ser a responsável por realizar a consulta, mas, sim, obter o resultado dela. Já a resposta para a segunda pergunta é **SIM**: geralmente outras classes também têm a necessidade de realizar uma consulta

pelo Id. Então para isso vamos trazer o nosso primeiro padrão de projeto: **Repository**.

6.1 Delegue o acesso a dados para classes Repositories

Esse padrão de projeto (*Design Pattern*) foi cunhado por Eric Evans e Martin Fowler e documentado por Eric Evans no livro *Domain-Driven Design*, em 2003. Ele consiste em prover uma abstração para acesso a dados. Veja a seguir a definição dada por esses e outros autores:

> "*Repositório é um mecanismo para encapsular armazenamento, recuperação e comportamento de pesquisa, que emula uma coleção de objetos.*" — Eric Evans, autor do livro *Domain-Driven Design*
>
> "*Mediador entre o domínio e as camadas de mapeamento de dados usando uma interface semelhante a uma coleção para acessar objetos de domínio.*" — Edward Hieatt e Robert Mee, no livro *Padrões de Arquitetura de Aplicações Corporativas*

Note que, em ambas as definições, é mencionado o termo "coleção" e que, através do Repository, vamos concentrar todo o acesso ao banco de dados. Ou seja, todas as consultas `SOQL` e operações de DML (`insert` , `update` , `delete`) devem ser realizadas através do Repository. Dessa forma, podemos mover as consultas soltas nos códigos para classes do tipo Repository. Geralmente, teremos uma classe Repository por `SObject` , então teremos `AccountRepository` , `ContractRepository` , `OpportunityRepository` e assim por diante. Vamos ver a implementação do `ContractRepository` :

```
public class ContractRepository {

    public List<Contract> findByIds ( List<String> ids ) {

        return [ SELECT Id, Status
                 FROM Contract
                 WHERE Id IN: ids ];

    }

    public List<Contract> findByAccountAndStatus
                          ( String accountId, String status ) {

        return [ SELECT Id, Status
                 FROM Contract
                 WHERE AccountId =: accountId
                    AND Status =: status ];

    }

    public List<Contract> save (List<Contract> contracts) {

        upsert contracts;

        return contracts;

    }

}
```

#se-liga-na-info 😎

Pontos de atenção na implementação de uma classe Repository

- Ela deve prover listas de objetos, de forma simples e direta;
- Utilize sempre `findByXXX` como nome de métodos de pesquisa;
- Toda transformação de dados e/ou manipulação deve ser feita fora da classe `Repository`, ou seja, ela recebe, persiste e retorna o dado, ou ela recebe informações, pesquisa e retorna uma lista de dados;
- Os métodos devem sempre retornar uma lista ou uma instância de um objeto. Já falei isso anteriormente, mas fica o reforço, só para você não esquecer. :)

Com isso, nosso código fica um pouco mais organizado, mas ainda longe do ideal. Vamos refatorá-lo para utilizar o `ContractRepository`:

```
List<Contract> amendmentContracts = new List<Contract> ();

// Determina quais são os contratos
// que foram alterados para Assinado

for ( Contract contract : newContracts ) {

    Contract oldContract = oldContracts.get ( contract.Id );

    if ( contract.Status == 'Assinado'
        && oldContract.Status != 'Assinado'
        && contract.OriginalContract__c != null ) {

        amendmentContracts.add ( contract );

    }
```

```
}

// cria uma lista de ids de contratos originais
List<String> originalContractIds = new List<String> ();

for (Contract contract : amendmentContracts) {
    originalContractIds.add ( contract.OriginalContract__c );
}

ContractRepository contractRepository = new ContractRepository();

// consulta os contratos originais para Inativação

List<Contract> originalContracts
                    = contractRepository
                            .findByIds ( originalContractIds );

// determina a relação do contrato original
// com o contrato assinado

if ( !originalContracts.isEmpty() ) {

    Map<Id, Contract> indexedOriginalContracts =
                        new Map<Id, Contract>(originalContracts);

    // determina o contrato original com base no novo contrato
    // assinado e atualiza o contrato original como Inativado
    for (Contract amendmentContract : amendmentContracts ) {

        Contract originalContract =
            indexedOriginalContracts.get (
                        amendmentContract.OriginalContract__c);

        originalContract.Status = 'Inativado';

    }

    contractRepository.save ( originalContracts );
```

```
}
```

Inicialmente, parece que o código teve pouca mudança, isso porque ele ainda está fazendo muitas outras coisas, que geram **distração**, aumentando a **complexidade cognitiva**.

Vamos então dar mais um passo para melhorar o nosso código. Para isso, veremos mais uma estratégia que gosto de aplicar para remover distrações do código, que é a implementação de classes **Filter**.

6.2 Filtre, indexe e agrupe os dados

Filter é um padrão que tem a responsabilidade de encapsular a lógica de filtro, indexação, agrupamento de `SObjects` . Então vamos simplesmente mover a implementação desse tipo de comportamento para uma classe Filter, a `ContractFilter` :

```
public class ContractFilter {

    public List<Contract> byChangedToSignedStatus (
                            List<Contract> newContracts,
                            Map<Id, Contract> oldContracts ) {

        List<Contract> amendmentContracts
                                        = new List<Contract> ();

        for ( Contract contract : newContracts ) {

            Contract oldContract =
                                oldContracts.get (contract.Id);

            if ( contract.Status == 'Assinado'
                && oldContract.Status != 'Assinado'
                && contract.OriginalContract__c != null  ) {
```

```
                amendmentContracts.add ( contract );

            }

        }

        return amendmentContracts;

    }

    public List<String> extractOriginalContractIds (
                                    List<Contract> contracts) {

        List<String> originalContractIds = new List<String> ();

        for (Contract contract : contracts) {
            originalContractIds.add (
                            contract.OriginalContract__c);
        }

        return originalContractIds;
    }

    public Map<Id, Contract> indexById (
                                    List<Contract> contracts ) {

        return new Map<Id, Contract>(contracts);

    }

}
```

Agora vamos aplicar a `ContractFilter` no código e ver como ele fica:

```
ContractFilter filter = new ContractFilter ();

List<Contract> amendmentContracts =
        filter.byChangedToSignedStatus (
                            newContracts, oldContracts );
```

```
ContractRepository contractRepository
                                    = new ContractRepository ();

List<Contract> originalContracts
                     = contractRepository.findByIds (
                         filter.extractOriginalContractIds
                                     ( amendmentContracts ) );

// determina a relação do contrato original com o contrato assinado

if ( !originalContracts.isEmpty() ) {

    Map<Id, Contract> indexedOriginalContracts =
                         new Map<Id, Contract>(originalContracts);

    // determina o contrato original com base
    // no novo contrato assinado e
    // atualiza o contrato original como Inativado
    for (Contract amendmentContract : amendmentContracts ) {

        Contract originalContract =
            indexedOriginalContracts.get (
                         amendmentContract.OriginalContract__c);

        originalContract.Status = 'Inativado';

    }

    contractRepository.save ( originalContracts );

}
```

Perceba que o número de linhas desse código reduziu drasticamente, em aproximadamente 66%. Mas ainda não é o ideal, pois você ainda precisa ler o código todo para absorver a intenção, que é inativar os *Contratos* originais quando os *Aditivos* forem assinados.

Note que aplicando somente **a segregação de responsabilidade** já melhoramos e muito a leitura do código atual, ou seja, deixamos a nossa classe mais **coesa**. As ações de pesquisar no banco e filtrar os contratos assinados ainda ocorrem, mas no lugar correto, diminuindo a distração da operação principal.

#fica-dica-clean-code

Quando falamos de **coesão**, também podemos falar que estamos aplicando o **Princípio da Responsabilidade Única** (SRP), que determina que uma classe ou método deve ter uma única razão para mudar, e não múltiplas razões.

Mas ainda não acabou! Temos muito o que fazer para melhorar esse código, então vamos dar mais um passo. Como podemos expressar melhor a nossa intenção de *Inativar os Contratos originais após a assinatura dos Aditivos*? Para isso, veremos mais um padrão que costumo aplicar em meus projetos: o **Enricher**.

6.3 Enriqueça dados

O padrão ***Enricher*** tem a responsabilidade de encapsular todo o processo de enriquecimento de dados de um determinado objeto. Ele nada mais é do que um *Builder* (GOF), pois tem a mesma necessidade (construir/modificar a estrutura de objetos complexos). Apesar de ser uma variação do *Builder*, nada impede que seja utilizada uma composição de um *Enricher* com uma outra classe *Builder* para diminuir a responsabilidade dela, fazendo com que ela seja uma orquestradora do fluxo de enriquecimento.

Por uma questão semântica, procure utilizar o nome *Enricher* em um escopo de *Trigger*, pois ela poderia ser chamada de `Builder` ou

`Service` em outros contextos de aplicação. Veremos exemplos desse uso mais à frente.

Para o nosso cenário, o `InactivateOriginalContractEnricher` poderia até se chamar de `InactivateOriginalContractService`, que ainda assim estaria desempenhando o papel de enriquecer o *Contrato* original. Perceba que o nome da classe expressa a sua intenção. Quando olhar no código, você vai compreender o propósito da classe para depois avaliar a sua execução:

```
public class InactivateOriginalContractEnricher {

    ContractRepository repository;

    ContractFilter filter;

    public InactivateOriginalContractEnricher () {
        this.repository = new ContractRepository ();
        this.filter = new ContractFilter ();
    }

    public List<Contract> inactivatedBy (
                        List<Contract> signedContracts ) {

        List<Contract> originalContracts =
                    findOriginalContracts ( signedContracts );

        if (originalContracts.isEmpty()) {
            return new List<Contract> ();
        }

        List<Contract> inactivatedContracts =
                inactivateOriginalContracts ( originalContracts
                                        , signedContracts );

        return repository.save ( inactivatedContracts );

    }
```

```
    private List<Contract> findOriginalContracts (
                          List<Contract> signedContracts ) {

        List<String> originalContractIds =
                        filter.extractOriginalContractIds (
                                              signedContracts );

        return repository.findByIds ( originalContractIds );

    }

    private List<Contract> inactivateOriginalContracts (
                        List<Contract> originalContracts,
                        List<Contract> signedContracts ) {


        Map<Id, Contract> indexedOriginalContracts =
                      this.filter.indexById (originalContracts);

        for (Contract signedContract : signedContracts ) {

            Contract originalContract =
                    indexedOriginalContracts.get (
                        signedContract.OriginalContract__c);

            originalContract.Status = 'Inativado';

        }

        return originalContracts;

    }

}
```

Agora vamos aplicar o nosso *Enricher* ao código contido dentro da *Trigger*:

```
ContractFilter filter = new ContractFilter ();
```

```
InactivateOriginalContractEnricher
            inactivateOriginalContract =
                 new InactivateOriginalContractEnricher ();

List<Contract> amendmentContracts =
        filter.byChangedToSignedStatus (
                             newContracts,oldContracts );

inactivateOriginalContract.inactivatedBy ( amendmentContracts );
```

O código anterior ficou muito mais direto e simplificado, e não aplicamos nenhuma técnica avançada de Orientação a Objetos, simplesmente aplicamos o **Princípio da Responsabilidade Única** (SRP), encapsulando sua responsabilidade. Como validamos esse princípio? Para responder isso, vamos nos fazer outras perguntas e, no final, chegaremos a uma conclusão.

Se mudarmos a regra de determinação dos aditivos que foram assinados, qual será o impacto? Será somente no método `byChangedToSignedStatus`.

Se mudarmos a regra para obtenção dos contratos originais, qual será o impacto? Será no método `findOriginalContracts` e possivelmente necessitará de um novo método na `ContractRepository`. Mesmo que tenhamos de fazer mais de uma modificação, ainda assim o impacto é baixíssimo.

Se mudarmos a regra de inativação de contratos originais, qual será o impacto? Somente o método `inactivateOriginalContracts` será alterado.

Agora reflita e responda à pergunta original: será que validamos a aplicação do **Princípio da Responsabilidade Única**?

#fica-dica-clean-code

O **Princípio da Responsabilidade Única** é o pilar fundamental do **Clean Code**. Sempre observe se sua classe/método está aderente a esse princípio. No começo será absurdamente complexo para você identificar, mas ele é uma busca constante e requer treino e autopoliciamento.

Os códigos que vimos aplicaram várias práticas do **Clean Code**, como manter o mesmo nível de atração por método, nomes significativos, ordem de declaração dos métodos. Veja a seguir essas práticas de forma mais detalhada.

Nomes significativos

Nomear é uma tarefa árdua, e de fato não é fácil. É justamente esse ponto que todo iniciante ignora, o que até faz sentido inicialmente, pois a intenção é produzir algo, independente se aquilo será evoluído ou não.

Mas percebo que, frequentemente, esse ponto, apesar de tolerável quando estamos no início do aprendizado, continua a ser deixado de lado e passa a ser uma prática, um hábito. Ocorre que um hábito de dar nomes totalmente irrelevantes para o contexto do algoritmo ou aplicação torna o código mais complexo de ser compreendido e evoluído. Como disse anteriormente, essa tarefa não é fácil, então vou trazer algumas dicas para facilitá-la.

Não utilize sufixo ou prefixos

Devs iniciantes costumam nomear variáveis conforme o seu tipo. Veja a seguir os **contraexemplos**:

```
Decimal totPrice = 1210.12;

String strName = '';
```

```
Date dtToday = Date.today();

Date myDate = Date.today(); // Data de quem ?

List<Order> lstOrder = new List<Order>();

List<Order> orderList = new List<Order>();

Map<String, Order> orderMap = new Map<String, Order>();

Map<String, Order> mapOrder = new Map<String, Order>();
```

Para coleções `List`, `Set` e `Map`, utilize somente o plural. Vamos renomear aplicando o plural e o contexto no qual elas estão envolvidas:

```
List<Order> orders = new List<Order>();

// Ordens alteradas
List<Order> changedOrders = new List<Order>();

// Ordens aprovadas
List<Order> approvedOrders = new List<Order>();

Map<String, Order> orders = new Map<String, Order>();

// Ordens Indexadas por Status
Map<String, Order> ordersByStatus = new Map<String, Order>();


// Ordens agrupadas por Status
Map<String, List<Order>> groupedOrdersByStatus
                           = new Map<String, List<Order>>();
```

Observe que, para variáveis do tipo `Map`, indicamos que temos um conteúdo indexado ou um agrupamento, mas em nenhum momento houve a necessidade de aplicar o tipo.

Para variáveis de outros tipos, utilize o contexto. Prefira substantivos e não verbos; não crie codificação; tente ser o mais explícito(a) possível.

```
Decimal totalAmount = 0.0;

Decimal price = 0.0;

String name = '';

Date today = Date.today();

Date createdDate = Date.today();

Boolean active = true;

Boolean closed = true;
```

Ordem de declaração de métodos

Pense em seu código como uma redação, que lemos de forma linear de cima para baixo. Então tente manter essa ordem de leitura, mantendo os métodos dependentes logo abaixo de sua utilização.

Como apresentado anteriormente, o método `inactivatedBy` está declarado logo ao topo, seguido de outros métodos, como `findOriginalContracts` e `inactivateOriginalContracts`, declarados nessa ordem. Observe novamente como estão declarados seguindo uma ordem de leitura de cima para baixo:

```
    public List<Contract> inactivatedBy (
                        List<Contract> signedContracts ) {

        List<Contract> originalContracts =
                    findOriginalContracts ( signedContracts );

        if (originalContracts.isEmpty()) {
            return new List<Contract> ();
        }
```

```
        List<Contract> inactivatedContracts =
                inactivateOriginalContracts ( originalContracts
                                            , signedContracts );

        repository.save ( inactivatedContracts );

    }

    private List<Contract> findOriginalContracts (
                            List<Contract> signedContracts ) {

        // conteúdo omitido

    }

    private List<Contract> inactivateOriginalContracts (
                        List<Contract> originalContracts,
                        List<Contract> signedContracts ) {


        // conteúdo omitido
    }
```

Mantenha o mesmo nível de abstração

Essa é uma prática bem interessante, pois promove uma facilidade na compreensão do código. Observe novamente o método `inactivatedBy` e perceba que ele basicamente utiliza outros métodos para executar o seu objetivo. Não temos lógica, a não ser verificar se há contratos para serem inativados. Ele mostra de forma clara e direta o seu objetivo. Caso tivesse mais lógicas e outros detalhes, isso distrairia você e tornaria o entendimento mais complexo.

```
public List<Contract> inactivatedBy (
                            List<Contract> signedContracts ) {

    List<Contract> originalContracts =
                findOriginalContracts ( signedContracts );
```

```
    if (originalContracts.isEmpty()) {
        return new List<Contract> ();
    }

    List<Contract> inactivatedContracts =
            inactivateOriginalContracts ( originalContracts
                                        , signedContracts );

    repository.save ( inactivatedContracts );

}
```

Para fechar o `#momento-clean-code`, posso dizer que essas práticas não serão adquiridas do dia para a noite, é um processo de adoção e atenção constante.

6.4 Encapsule o comportamento de validação com Validator

Esse é um outro exemplo de segregação de responsabilidade que podemos aplicar em vários lugares, mas vamos utilizá-los agora em *Trigger*. Basicamente, consiste em segregar as validações para uma classe que tenha somente este papel, o de **validar**.

> **#fica-dica-clean-code**
>
> Aqui, gosto de reforçar para que você não fique tentado(a) a aproveitar e fazer uma atribuição juntamente com o fluxo de validação, pois isso quebraria o **Princípio da Responsabilidade Única** (SRP).

Em nossa classe de *Trigger*, temos a necessidade de validar o CPF/CNPJ do *Cliente*. Já foi utilizada uma classe específica para validação, o que é excelente, mas não é o bastante, pois podemos ter várias outras validações que podem ocorrer em um Objeto Salesforce.

Veja novamente o código da *Trigger* com a implementação atual do `BrazilianDocumentValidator`:

```
/**
 * @author eduardo.bisso
 */
trigger Account on Account (before insert
                          , before update) {

    List<Account> newAccounts = Trigger.new;

    BrazilianDocumentValidator validator;

    switch on Trigger.operationType {

        when BEFORE_INSERT, BEFORE_UPDATE {

            for ( Account account : newAccounts ) {

                if ( !validator
                        .isValid(account.DocumentNumber__c) ) {

                    account
                        .DocumentNumber__c
                        .addError ('Documento Inválido' );

                }

            }

        }

    }
```

```
}
```

Para melhorar essa implementação, podemos segregar todo o fluxo de validação para uma outra classe, chamada `AccountValidator`, que será a responsável por realizar as validações de `Account`:

```
/**
 * @author eduardo.bisso
 */
public class AccountValidator {

    BrazilianDocumentValidator documentValidator;

    public AccountValidator () {
        this.documentValidator = new BrazilianDocumentValidator();
    }

    public void validate (List<Account> accounts) {

        for (Account account : accounts) {

           validateDocumentNumber(account);

        }

    }

    private void validateDocumentNumber (Account account) {

        if ( this.documentValidator.isValid (
                                account.DocumentNumber__c))  {
            return;
        }

        account.DocumentNumber__c
                        .addError ('Documento Inválido' );
    }
```

```
}
```

Perceba que a classe `AccountValidator` fornece somente um método para validar, ou seja, fornece somente um método público (`public`), e o restante é privado (`private`). É dessa forma que orientamos o consumo dessa classe. No caso, a *Trigger* `Account` só precisa se preocupar em chamar o `validate()` .

Outro ponto importante é a maneira como o método `validate()` foi construído: ele delega a responsabilidade de validar para cada método específico e, assim, ele fica mais legível e escalável.

> O exemplo anterior reforça a prática de **MANTER UM ÚNICO NÍVEL DE ABSTRAÇÃO POR MÉTODO**, fazendo com o que o método `validate` somente chame outros métodos. Dessa forma ele fica mais simples de entender e evoluir.

Chegou a hora de refatorar a *Trigger* para utilizar a nova implementação do `AccountValidator` :

```
/**
 * @author eduardo.bisso
 */
trigger Account on Account (before insert
                            , before update) {

    List<Account> newAccounts = Trigger.new;

    AccountValidator validator = new AccountValidator ();

    switch on Trigger.operationType {

        when BEFORE_INSERT, BEFORE_UPDATE {

            validator.validate (newAccounts);
```

```
        }

    }

}
```

Com essa implementação, a *Trigger* ficou mais enxuta, mas ainda apresenta problemas, que, para serem resolvidos, precisamos de mais conhecimento voltado à Orientação a Objetos, o que veremos no próximo capítulo.

Este capítulo trouxe várias práticas de **Clean Code** para você colocar em sua caixa de ferramentas. Sei que não é simples nem fácil aplicá-los de primeira, mas isso é uma questão de prática e, com ela, vem o hábito.

CAPÍTULO 7
Vamos adicionar novos Jutsus com a Orientação a Objetos

No capítulo anterior, vimos que somente aplicando o **Princípio da Responsabilidade Única** podemos melhorar absurdamente a qualidade do código que fazemos, isso porque ele fica mais bem distribuído e organizado, facilitando a compreensão e sua evolução.

Assim como o Naruto, que aprende novos *Jutsus* e técnicas para ficar mais forte, você também aprenderá novas características e técnicas que fará com que seu código fique mais forte!

Neste capítulo, veremos diversos conceitos da Orientação a Objetos que serão a base para a nossa próxima evolução, tais como sobrecarga, herança, polimorfismo e outros.

Para começar, veremos o conceito de **sobrecarga**, ou *overloading*. Para explicá-lo, vou trazer o exemplo do nosso `ContractRepository`, que contém uma operação de `save` que recebe uma `List` de `Contract`. E se quiséssemos ter um `save` que salvasse somente um `Contract`, é possível? Sim, e é isso que se chama **sobrecarga** de parâmetros de métodos.

7.1 Sobrecarga

Quando temos a necessidade de executar o mesmo método de forma diferente, podemos aplicar o conceito de **sobrecarga**, como no cenário do método `save`, onde desejamos salvar um contrato ou uma lista dele. A **sobrecarga** consiste em manter o mesmo nome de método e retorno, variando os parâmetros. Essa variação de parâmetros deve ser em número, tipo ou ordem.

Aplicando esse conceito, fornecemos alternativas de execução, porém mantendo a mesma natureza de operação. Dessa forma, melhoramos a expressividade e reusabilidade de nossas classes. Veja o cenário do método `save`, onde vamos fornecer alternativas de utilização:

```
public class ContractRepository {

    // métodos de find foram removidos para
    // facilitar a leitura

    public List<Contract> save ( List<Contract> contracts ) {

        upsert contracts;

        return contracts;

    }

    public Contract save ( Contract contract ) {

        upsert contract;

        return contract;

    }

}
```

Podemos implementar variações do código anterior. Um exemplo seria fornecer um `upsert` que recebe o campo que representa a chave externa:

```
public class ContractRepository {

    // métodos de find e os outros foram removidos para
    // facilitar a leitura
```

```
    public List<Contract> save ( List<Contract> contracts
                            , Schema.SObjectField field ) {

        upsert contracts field;

        return contracts;

    }

}
```

Quando tentamos salvar esse código, o compilador retorna o erro `Invalid field for upsert, must be an External Id custom or standard indexed field: field`. Esse erro está relacionado ao comando `upsert`, que necessita que seja informado o campo de forma explícita, não somente referência de um `Schema.SObjectField`. Vou aproveitar esse erro para apresentar a você a classe `Database`, que também fornece as operações de DML, com várias variações:

```
public class ContractRepository {

    // métodos de find e os outros foram removidos para
    // facilitar a leitura

    public List<Contract> save ( List<Contract> contracts
                            , Schema.SObjectField field ) {

        Database.upsert( contracts, field );

        return contracts;

    }

}
```

Podemos utilizar a classe `Database` para executar as operações DML, entre outras funcionalidades. Em nosso exemplo, utilizamos o método `upsert`, que recebe a lista de objetos que será persistida e o

campo que deverá ser utilizado como chave externa. Esse campo é definido através do tipo `Schema.SObjectField`.

A classe `Database` é um excelente exemplo de aplicação de sobrecarga, como no caso dos métodos que fazem operações DML.

Vamos analisar algumas das variações do método `upsert` da classe `Database`:

- `upsert ( record )`: recebe somente um objeto para persistir;
- `upsert ( records )`: recebe uma lista de objetos para persistir;
- `upsert ( record, externalIdField )`: recebe somente um objeto para persistir e o campo que será utilizado com chave externa;
- `upsert ( records, externalIdField )`: recebe uma lista de objetos para persistir e o campo que será utilizado com chave externa.

Para saber mais sobre a classe `Database`, acesse: https://developer.salesforce.com/docs/atlas.en-us.apexref.meta/apexref/apex_methods_system_database.htm.

Note como a própria API utiliza as vantagens de aplicação de sobrecarga para fornecer opções de execução para a classe cliente. Esse recurso também pode ser aplicado aos construtores de nossas classes.

Sobrecarga de construtores

Como comentado anteriormente, os construtores também podem ser sobrecarregados, ou seja, podemos ter mais de um construtor com o mesmo nome, porém com diferentes parâmetros.

Veja a seguir nossa classe que representa um Pedido. É através dela que receberemos dados do sistema da *Bisso Capital Inc.* para a venda de serviços. Nessa implementação, vamos utilizar um recurso que é chamado de `InnerClass`, uma classe interna, para trazer a representação de endereço (`Address`) e item do pedido (`Item`):

```
public class PurchaseOrder {

    String customerOrder;

    Account account;

    Address billingAddress;

    List<Item> items;

    public PurchaseOrder () {
        this.billingAddress = new Address();
        this.items = new List<Item>();
    }

    public class Address {
        // campos de address omitidos
    }

    public class Item {
        // campos de address omitidos
    }

}
```

Quando criamos classes, devemos pensar na melhor forma de como elas serão consumidas. Esse pensamento remete ao que vimos no capítulo de testes de unidade quando comentei sobre comportamento empático. Com isso em mente, como podemos fornecer alternativas de consumo da classe `PurchaseOrder` ?

No caso, a classe está fornecendo somente o construtor padrão, sem argumentos, forçando as classes clientes a fazerem diversas atribuições, como `account` , `billingAddress` , entre outros. Podemos facilitar esse processo de criação da instância da classe através da **sobrecarga de construtores**. Veja o código a seguir, onde fornecemos opções para construção de sua instância:

```
public class PurchaseOrder {

    // campos de PurchaseOrder omitidos

    public PurchaseOrder () {
        this.billingAddress = new Address();
        this.items = new List<Item>();
    }

    public PurchaseOrder (Account account) {
        this();
        this.account = account;
    }

    public PurchaseOrder (Account account, Address billingAddress) {
        this(account);
        this.billingAddress = billingAddress;
    }

}
```

Observe que nessa implementação temos diversos construtores sobrecarregados, e em cada sobrecarga utilizamos o comando `this(...)`. Essa é a forma como reutilizamos chamadas de construtores. Mas vale lembrar que não somos obrigados a executar sempre a chamada do `this(...)`, devemos usar somente quando temos a intenção de reaproveitar outro construtor.

#se-liga-na-info 😎

O comando `this(...)`, quando mencionado, deve ser o primeiro comando do construtor. Veja o contraexemplo a seguir:

```
public class PurchaseOrder {

   public PurchaseOrder () {
       //código omitido
   }

   public PurchaseOrder (Account account) {
       this.account = account;
       this(); // erro de compilação
   }

}
```

Armadilhas da sobrecarga

Veremos agora diversas dicas que você pode aplicar no seu dia a dia para evitar cair em armadilhas de código que podem ser causadas com a utilização de **sobrecarga** de forma indevida.

#fica-dica-clean-code

A regra dos três parâmetros

Mais uma **dica** de *Clean Code* que vai ajudar a deixar o seu código "bonito e cheiroso" é o cuidado com o excesso de parâmetros que utilizamos em nossos códigos. Quando temos muitos parâmetros, o código fica mais complexo de ser entendido. Recomendo que utilize no máximo 3 (três) parâmetros em um método ou construtor.

Vamos ver alguns exemplos com a nossa classe `PurchaseOrder` que criamos anteriormente. Nós fizemos uma sobrecarga de construtores, passando o `Account` como parâmetro; agora vamos aumentar esse volume de parâmetros, passando os campos de `Address`:

```
public class PurchaseOrder {

    // campos de PurchaseOrder e outros
    // construtores foram omitidos

    public PurchaseOrder (Account account) {
        this();
        this.account = account;
    }

    public PurchaseOrder (Account account,
                          String street,
                          String city,
                          String state,
                          String zipCode) {

        this();
        this.account = account;
```

```
        this.billingAddress = new Address(street,
                                          city,
                                          state,
                                          zipCode);
    }

}
```

Como seria o consumo dessa classe? Vou fazer questão de deixar o conteúdo de cada variável menos claro, para que você tente olhá-lo de forma rápida, sem colar. Tente identificar qual parâmetro corresponde a qual atributo:

```
Account account = new Account ();

PurchaseOrder purchaseOrder = new PurchaseOrder ( account ,
                                  'DELHOURME',
                                  'CAUDOS',
                                  'LIBOURNE',
                                  'ABC');
```

Quando temos vários parâmetros do tipo `String` seguidos, temos mais dificuldade em determinar qual parâmetro representa o quê. Será que o terceiro parâmetro é a cidade, o estado ou o CEP? Para identificá-los, você precisa contar com a IDE ou voltar no código para ver a assinatura do método ou construtor.

Mas o número três, que eu sugeri como máximo de parâmetros, é mágico? Não, e não quer dizer que seu código ficará ruim porque teve a necessidade de utilizar mais do que três parâmetros. Ele só ficará mais difícil de ser consumido e evoluído.

Quando tiver a necessidade de passar diversos parâmetros para um método/construtor, opte por encapsulá-los em uma classe. Dessa forma, fica mais claro o que precisa ser informado.

#fica-a-dica 🤔

O excesso de parâmetros em um método ou construtor não prejudica a classe em si, mas sim aquela que a consome. *A empatia no código está na forma como o consumimos*.

Uma outra forma que pode ajudar a deixar o código mais elegante é utilização do padrão *Builder* com *fluent interface*. Despertei sua curiosidade? Esse padrão será apresentado mais à frente, mas vou dar um *spoiler*: observe a mesma implementação utilizando esse padrão.

```
Account account = new Account ();

PurchaseOrder purchaseOrder = new PurchaseOrder ( account ,
                                  new BillingAddress()
                                      .withStreet('DELHOURME')
                                      .withCity('CAUDOS')
                                      .withState('LIBOURNE')
                                      .withZipCode('ABC') )
```

Aposto que você conseguiu identificar com clareza qual parâmetro está sendo passado e qual a intenção do método.

Não utilizar parâmetros `Boolean`

Evite ao máximo utilizar parâmetros `Boolean`, pois existe uma enorme tendência de que o método ou construtor esteja tomando uma decisão com base nesses parâmetros. A classe que fornece o método/construtor delega a tomada de decisão para uma classe terceira, o que deixa o consumo obscuro e aumenta a complexidade cognitiva.

Observe o exemplo a seguir do método que tem o objetivo de calcular o preço de um item e retornar uma instância do objeto `CartItem`:

```
PricingService service = new PriceService();

CartItem cartItem = service.checkPriceAndAvailability ( customer
                                                      , product
                                                      , true );
```

Perceba como é difícil determinar o propósito do parâmetro com valor `true`. Somente olhando para o método, não dá para identificar qual o objetivo desse parâmetro. Então você precisa entrar no método para identificar que ele utiliza essa variável para determinar a quantidade disponível considerando estoque especial.

Esse é o cenário onde caberia um outro método mais específico para expressar esse comportamento, como `checkPriceAndAvailabilityWithSpecialStock`:

```
PricingService service = new PriceService();

CartItem cartItem
            = service
               .checkPriceAndAvailabilityWithSpecialStock (
                                          customer
                                        , product );
```

Esse trecho de código deixa clara a nossa intenção sem a necessidade de entrar na classe para saber o que ela vai fazer.

#fica-a-dica 🤔

Você pode estar se perguntando: "Mas, *Bisso*, e se eu precisar? Me obrigaram! Tem uma forma, mais elegante, de utilizar?"

Se não tem outro jeito, colocaram bambu embaixo da sua unha :). Você pode utilizar uma alternativa simples e que vai ajudar um pouco a expressividade do seu código. Ela consiste em criar uma variável do tipo `Boolean` que represente a intenção. Veja o exemplo a seguir:

```
PricingService service = new PriceService();

Boolean usingSpecialStock = true;

CartItem cartItem = service.checkPriceAndAvailability ( customer
                                        , product
                                        , usingSpecialStock );
```

Só o fato de passar uma variável indicando o contexto da execução melhora muito a expressividade do código. Mas lembre-se de utilizar essa técnica como última alternativa.

Sobrecarga de construtores em SObjects

Quando apresentei os `SObjects`, mencionei que eles são classes especiais para Salesforce que trazem a capacidade de utilizar construtores com parâmetros nomeados, o que podemos assumir aqui como uma forma de sobrecarga de construtores. Vamos ver como ele é interessante para nossa implementação.

```
Account account = new Account (
                        Name = 'Bisso Capital INC',
                        BillingStreet = 'Rua dos Bissos 124',
                        BillingCity = 'São Paulo' );
```

Esse recurso permite que você informe os campos de forma clara e direta durante a criação da instância. Procure utilizar esse recurso em mapeamentos simples e diretos, não realizando operações na sua construção. Veja o contraexemplo a seguir:

```
Opportunity opportunity = new Opportunity (
    AccountId = ( opportunityRequest.accountId != null )
                    ? opportunityRequest.accountId : null ,
    Lead__c =   ( opportunityRequest.leadId != null )
                    ? opportunityRequest.leadId : null
);
```

Esse contraexemplo traz diversas distrações, e temos que ler cada instrução para determinar o comportamento do código. Evite implementações como essa!

Repetição de código pode, Arnaldo?

Um dos maiores problemas no desenvolvimento de software é a repetição de código, que, no livro *Clean Code*, é representada pelo princípio *DRY*, *Don't Repeat Yourself*, que, em português, podemos chamar **Não se repita**.

Quando somos inexperientes, tentamos evitar a repetição de código incluindo novas condições em nossos algoritmos, através de um conjunto de `if` ou `switch`. Porém, essa tentativa pode elevar a complexidade e diminuir a manutenibilidade, surtindo o efeito contrário da nossa intenção.

Para atacar esse problema, vamos nos debruçar sobre mais conceitos de Orientação a Objetos, como **herança**, **polimorfismo** e outros. Eles são os **eliminadores de** `if`, para deixar seu código mais elegante e mais reutilizável.

Vamos pensar em classes do tipo `Repository` que fazem acesso ao banco e realizam operações de CRUD, como `save`, `remove`, entre outras. Se tivéssemos de implementar uma `AccountRepository` ou

`ContactRepository`, provavelmente elas teriam os métodos de `save` e `remove` iguais e estaríamos **repetindo código**.

Mas todo `AccountRepository`, `ContractRepository`, `ContactRepository` **é um** `Repository`, não é? Sim, esse é o conceito de **herança** na Orientação a Objetos, quando uma classe pode adquirir características e comportamentos de outra classe, assim como acontece na vida real. Veja a seguir como podemos utilizar esse conceito para aumentar o reúso de código e **evitar repetição**.

7.2 Herança

Como já comentei com você, a Orientação a Objetos visa representar o mundo real em software, e um dos pontos é a nossa capacidade de **herdar** geneticamente características e comportamentos de nossos pais. A linguagem APEX também traz essa capacidade: uma classe pode **herdar** características de comportamentos de outra classe.

A plataforma Salesforce utiliza o conceito de **herança** em todas as classes que representam um Objeto Salesforce, como `Account`, `Contact`, `Contract`, entre outros, que são **filhos** da classe `SObject`. A classe `SObject` é a classe-base de todos os objetos. Com isso, eles herdam as características e comportamentos de sua classe-mãe. Por exemplo, o método de `addError` utilizado em nossa implementação de *Trigger* vem da classe-mãe `SObject`.

```
Account account = new Account( Name = 'Teste do Bisso');

account.addError('Nome inválido');
```

Agora veja essa hierarquia de forma gráfica:

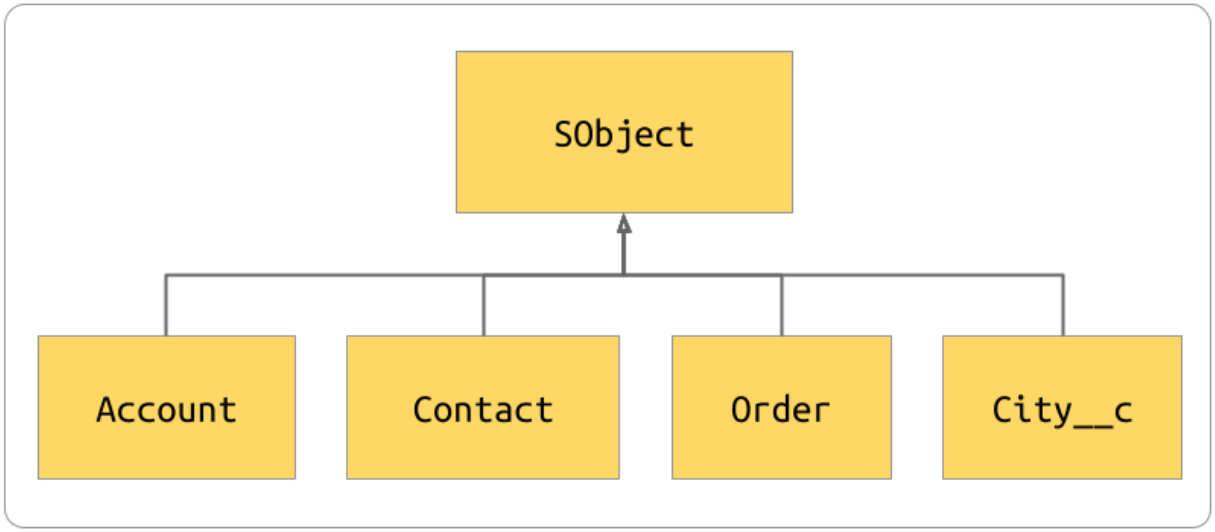


Figura 7.5: Filhas de SObject

A classe `SObject` é representação básica de um Objeto Salesforce, mas, quando tratamos de qualquer classe APEX, a linguagem nos traz a classe `Object` como a mãe de todas as classes. A classe `Object`, oriunda do Java, fornece os métodos `toString`, `equals`, `hashCode`, entre outros, para todas as classes existentes na linguagem. Então vamos corrigir o diagrama apresentado anteriormente:

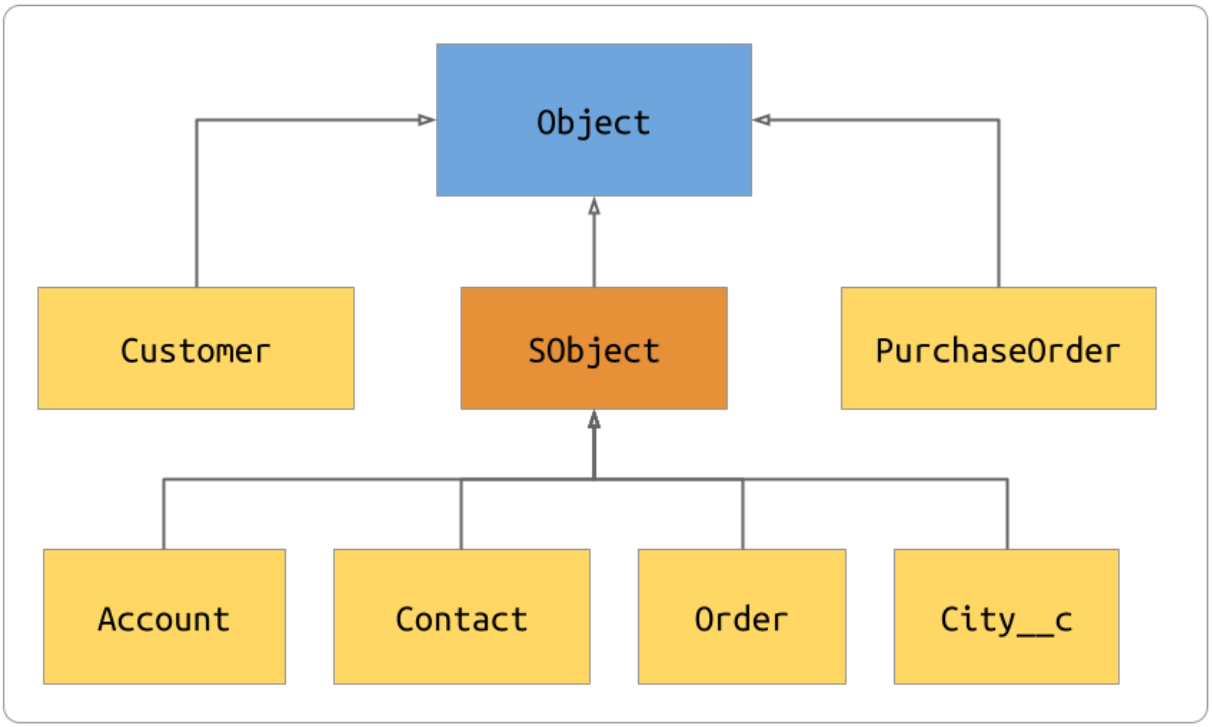


Figura 7.6: Hierarquia completa com a classe Object

Note que agora teremos outras classes participando: `Customer` e `PurchaseOrder`. Elas não são `SObject`, mas são `Object`. Nesse exemplo, podemos ver que `Account` é neta de `Object`, então também **é um** `Object`, e as outras classes, como `Customer` e `PurchaseOrder`, são somente `Object`.

Mas como faço para herdar uma classe? Utilizamos a palavra reservada `extends` e a palavra reservada `virtual`.

Vamos ver um exemplo, que é a representação dos funcionários de nossa financeira, a *Bisso Capital Inc.*. Nela, nós temos Analistas Financeiros, Gerentes de Contas, Diretores, Assessores de Investimento. Então vamos criar as classes `Advisor`, que é a representação do *Assessor*, e a `Manager`, que é a representação do *Gerente de Contas*:

```
public class Advisor {

    public String code;
```

```
    public String name;

    public String documentId;

    public Decimal salary;

    public List<Account> accounts;

}

public class Manager {

    public String code;

    public String name;

    public String documentId;

    public Decimal salary;

    public List<Advisor> advisors;

}
```

Vamos testar essas classes e avaliar a execução:

```
@isTest
public class AdvisorAndManagerTest {

    @isTest
    public static void testAdvisorAndManager () {

        Advisor advisor = new Advisor();
        advisor.code = '123';
        advisor.name = 'Bisso';
        advisor.documentId = '123456789';
        advisor.salary = 5000;
        advisor.accounts = new List<Account>();
```

```
        System.debug ( JSON.serialize ( advisor ) );

        Manager manager = new Manager();
        manager.code = '344';
        manager.name = 'Super Manager';
        manager.documentId = '982338233';
        manager.salary = 15000;
        manager.advisors = new List<Advisor>();

        System.debug ( JSON.serialize ( manager ) );

    }

}
```

No cenário apresentado, estamos literalmente duplicando código. Se observarmos as características de ambas as classes, podemos ver que várias delas são informações básicas de um funcionário.

Nesse momento, começa o nosso trabalho de encapsulamento de características e comportamentos. Podemos aplicar o conceito de herança com a intenção de fornecer uma estrutura básica de um funcionário, através da classe `Employee`. Ela será a representação do funcionário, dado que todos têm um código, nome, documento de identificação e salário.

```
public class Employee {

    public String code;

    public String name;

    public String documentId;

    public Decimal salary;

}
```

Agora que temos todas as características comuns de um funcionário em `Employee`, vamos fazer com que a classe `Advisor` herde suas características utilizando a palavra reservada `extends`:

```
public class Advisor extends Employee {

    // todas as características comuns estão em
    // Employee

    public List<Account> accounts;

}
```

No entanto, ao tentarmos compilar o código anterior, recebemos o erro `Non-virtual and non-abstract type cannot be extended: Employee (1:14)`, que diz que a classe `Employee` não pode ser herdada, por não ser virtual ou abstrata. Esse erro ocorre porque, no `APEX`, para que uma classe possa ser estendida, ela deve conter o modificador `virtual`. Então vamos corrigir a `Employee` e compilar novamente o código da `Advisor`:

```
public virtual class Employee {

    // campos omitidos

}
```

Após compilar a `Employee`, podemos compilar novamente a `Advisor` e ver que não ocorrem mais erros. Com isso, podemos aplicar o código às outras classes, que agora passam a herdar de `Employee`:

```
public class Manager extends Employee {

    public List<Advisor> advisors;

}

public class Analyst extends Employee {
```

```
    public List<Asset> assets;

}

public class Director extends Employee {

}
```

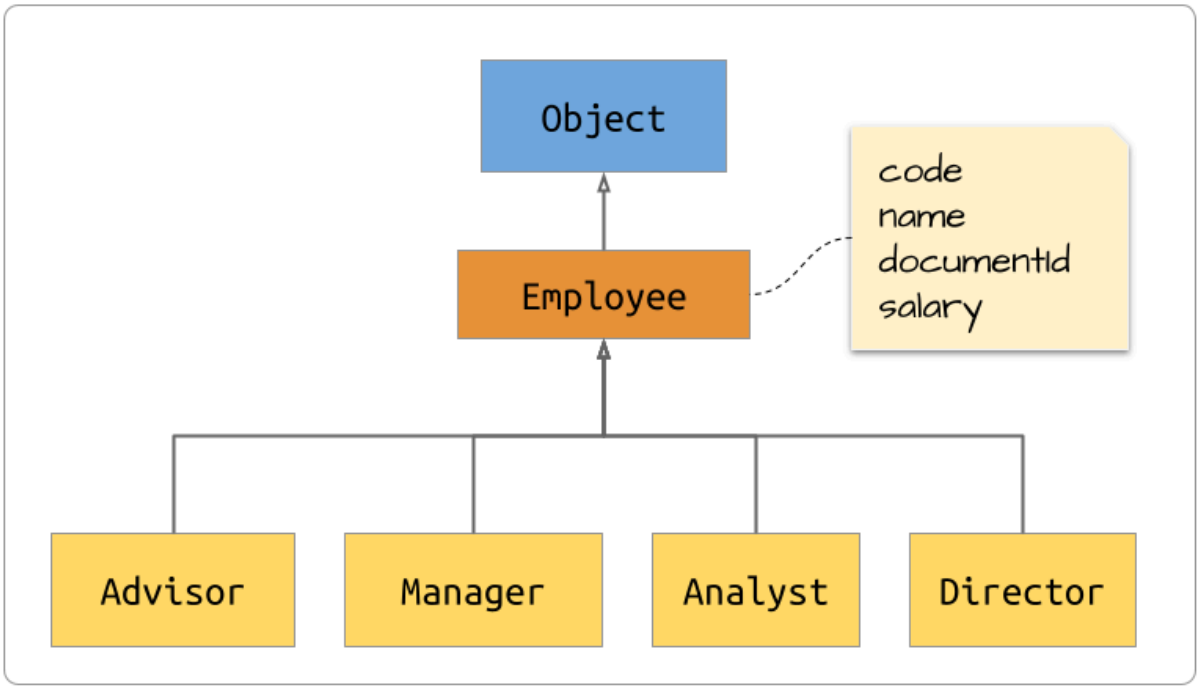


Figura 7.7: Nossa primeira visão das classes filhas de Employee

Note que **não há mais repetição de código**. Cada classe ficou elegante e cada uma possui somente suas características específicas para o tipo do funcionário. Volte à classe de teste `AdvisorAndManagerTest`, compile e execute novamente o teste. Veja que nada mudou, o código permanece o mesmo, sem erros, e o resultado também é o mesmo.

Até aqui vimos os primeiros passos para herdar uma classe e como suas características são reutilizadas pela classe filha. Agora acompanhe a saga da *Bisso Capital Inc.* para ver novos recursos que podemos obter utilizando **herança**.

O time de desenvolvimento da *Bisso Capital Inc.* recebeu uma nova demanda de atribuição de um bônus a todos os funcionários da empresa, que é o salário mais 5%. Para atender a essa demanda, vamos nos atentar ao trecho da frase: *Um bônus a todos os funcionários.*

Esse comportamento poderia ser aplicado à classe `Employee`, que representa os funcionários; dessa forma, todos os outros tipos de funcionários herdarão o mesmo bônus:

```
public virtual class Employee {

    // campos omitidos

    public Decimal getBonus () {
        return this.salary * 1.05;
    }

}
```

Agora vamos criar uma classe para testar todas as nossas funcionalidades voltadas aos funcionários:

```
@isTest
public class EmployeeTest {

    @isTest
    public static void testEmployees () {

        Advisor advisor = new Advisor();
        advisor.salary = 5000;
        Decimal advisorBonus = advisor.getBonus();

        System.assertEquals ( 5250, advisorBonus );

        Manager manager = new Manager();
        manager.salary = 15000;
        Decimal managerBonus = manager.getBonus();
```

```
        System.assertEquals ( 15750, managerBonus );

        Analyst analyst = new Analyst();
        analyst.salary = 10000;
        Decimal analystBonus = analyst.getBonus();

        System.assertEquals ( 10500, analystBonus );
    }

}
```

Sucesso! Funcionalidade entregue, cliente satisfeito! E o código **não tem um monte de** `ifs` , está lindo com apenas três linhas de código. Mas você acha que parou nesse requisito?

Como diria *Joseph Climber*, a vida é uma caixinha de surpresas! Com a aplicação da política de bônus, os **Analistas de Investimento** ficaram insatisfeitos com o valor recebido e entraram em greve. Eles alegaram que a *Bisso Capital Inc.* só obteve lucros extraordinários graças ao seu trabalho, de análise e conhecimento do mercado financeiro.

Com isso, a *Bisso Capital Inc.* alterou a regra de bonificação dos **Analistas de Investimento**, adicionando mais 7 mil reais (R$ 7.000,00) à bonificação recebida de todo funcionário. E agora, como tratamos isso? Colocamos `ifs` no método `getBonus` ? Não! Vamos dar mais um passo em nosso conhecimento de Orientação a Objetos, redeterminando o comportamento da classe-mãe, utilizando sobrescrita de métodos (*Overriding*).

Sobrescrita de métodos

A sobrescrita de métodos é uma das capacidades que obtemos através da herança, dessa forma a classe filha pode redeterminar o comportamento da classe-mãe. Ela pode ocorrer em qualquer método visível da hierarquia de classes, ou seja, não se aplica somente à classe-mãe, mas, sim, a qualquer classe da árvore de herança.

Para isso, precisamos voltar à classe `Employee` e adicionar o modificador `virtual` ao método `getBonus`. Isso permitirá que outras classes possam sobrescrever o método, ou seja, realizar o *override*:

```
public virtual class Employee {

    // campos omitidos

    virtual
    public Decimal getBonus () {
        return this.salary * 1.05;
    }

}
```

Após habilitar o método para ser sobrescrito, vamos criar o novo método `getBonus` em `Analyst` com a mesma assinatura do método `getBonus` de `Employee`. O novo método também deve estar anotado com `override`, indicando que ele é um método sobrescrito:

```
public class Analyst extends Employee {

    // todas as características foram omitidas

    override
    public Decimal getBonus() {
        return this.salary * 1.05 + 7000;
    }

}
```

Métodos sobrescritos possuem algumas regras:

- O método da **classe-mãe** deve ser `virtual`;
- Deve ser mantida a mesma assinatura da **classe-mãe**, e isso quer dizer que o tipo de retorno, nome e parâmetros devem ser o mesmo, e não posso ter uma visão mais restritiva do que a **classe-mãe**;

- O método sobrescrito deve ter o modificador/anotação `override`.

Agora que ajustamos a implementação da classe `Analyst`, vamos executar o nosso teste, que deve dar erro:

```
System.AssertException: Assertion Failed:
Expected: 10500, Actual: 17500.00

Class.EmployeeTest.testEmployees: line 26, column 1
```

O teste deveria falhar mesmo! Então vamos corrigi-lo com o novo cenário:

```
@isTest
public class EmployeeTest {

    @isTest
    public static void testEmployees () {

        // código omitido

        Analyst analyst = new Analyst();
        analyst.salary = 10000;
        Decimal analystBonus = analyst.getBonus();

        System.assertEquals ( 17500, analystBonus );

    }

}
```

Devemos estar sempre atentos à repetição de código, o princípio **DRY**. Estamos repetindo a regra de `Employee` em `Analyst`. Como resolvemos isso? Simples: utilizando a palavra reservada `super` para chamar o método da **classe-mãe**.

```
public class Analyst extends Employee {

    // todas as características foram omitidas
```

```
    override
    public Decimal getBonus() {
        return super.getBonus()  + 7000;
    }

}
```

Através da palavra reservada `super`, acessamos os membros da classe-mãe, ou seja, o método `getBonus` da classe `Employee`.

A sobrescrita é um recurso extremamente poderoso da Orientação a Objetos, e diversos padrões de projeto utilizam esse recurso como parte de sua estrutura.

Herança e ciclo de vida dos objetos

Agora que vimos vários conceitos da Orientação a Objetos, vamos ver como esses objetos que utilizam herança se comportam na memória *heap*.

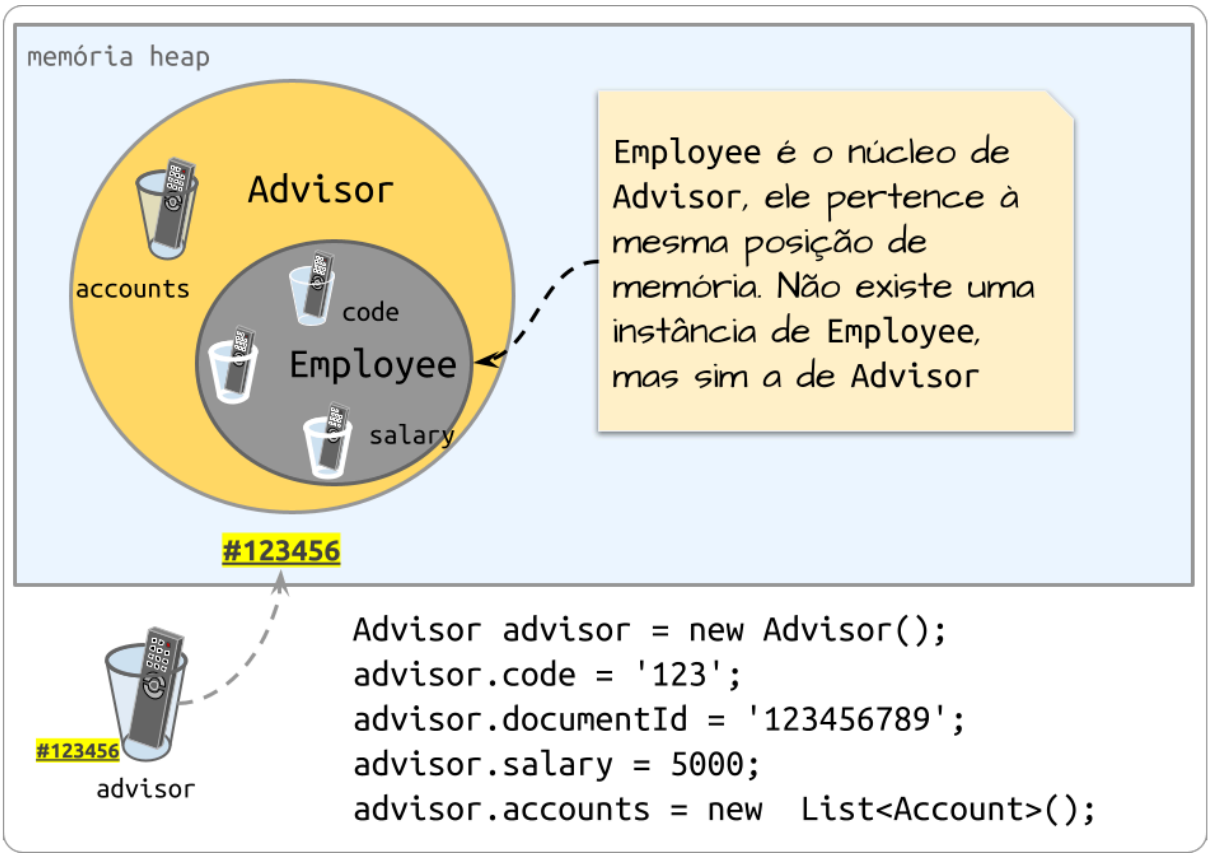


Figura 7.8: Representação de Advisor na memória

Note na imagem anterior que a instância de `Advisor` possui uma espécie de núcleo, que é a parte `Employee` de `Advisor`. Ele contém as características e comportamentos da classe `Employee`, que agora fazem parte da `Advisor`.

Quando realizamos atribuições na instância de `Advisor` aos atributos `code`, `salary`, estamos alterando a parte `Employee` de `Advisor`, porém ainda continua sendo um `Advisor`.

Ciclo de inicialização de objetos

Através do construtor, criamos as instâncias dos objetos. Como fica o seu ciclo de vida quando utilizamos herança?

Como vimos, `Employee` é o núcleo de `Advisor`, então quando realizamos a criação de uma nova instância de `Advisor`, `new Advisor();`, é realizada uma sequência de operações:

1. Começa a execução do construtor de `Advisor`;
2. Realiza a construção de seu núcleo, `Employee`;
3. Finaliza a construção de `Advisor`.

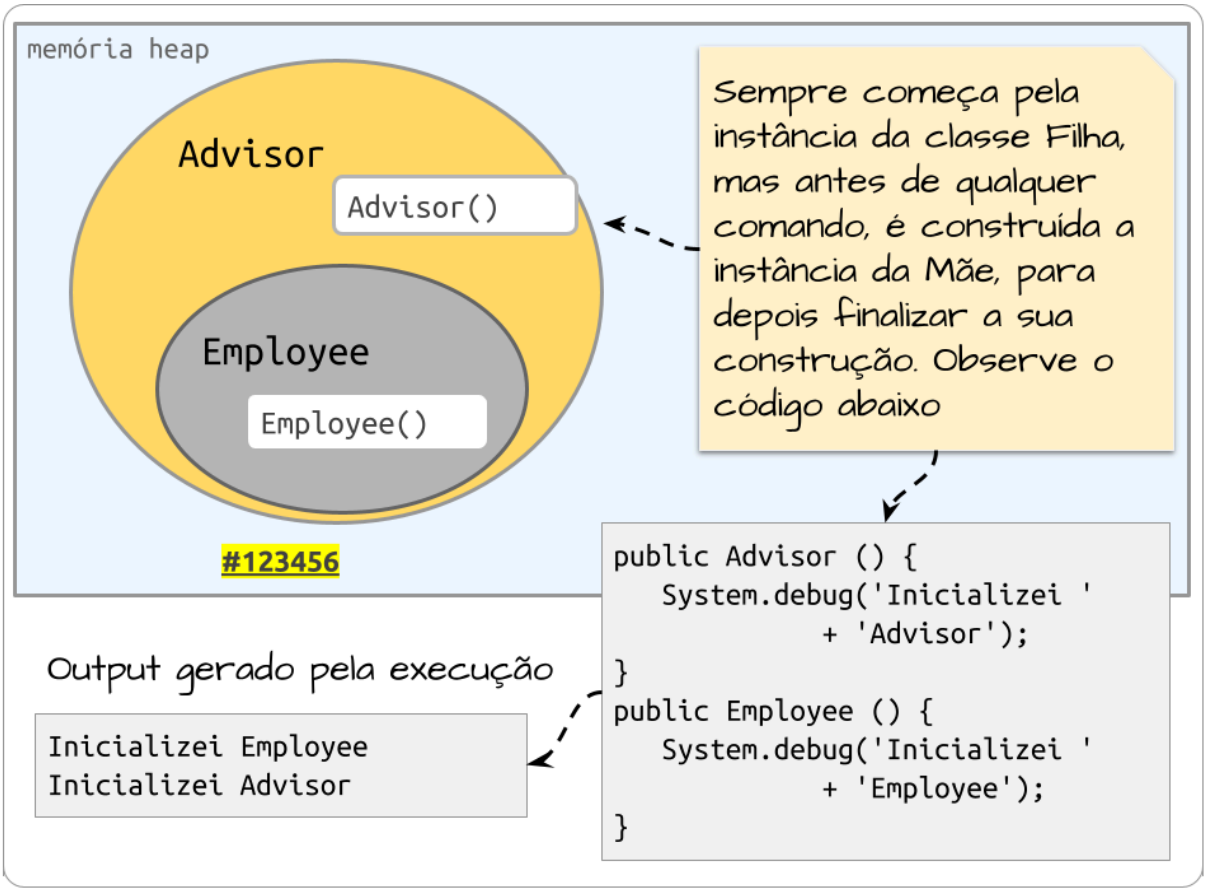


Figura 7.9: Ciclo da construção de objetos

Esse fluxo é executado através da execução implícita do construtor da superclasse, a **classe-mãe**. Quando executamos a construção de `Advisor`, o compilador buscará o construtor `default` (*padrão*) da classe-mãe, através do comando `super()`. Esse processo só é possível quando a classe-mãe possui um construtor `default`, um construtor sem argumentos.

Observe no próximo bloco de código que deixamos a chamada `super();` explícita no código, que é o que aconteceu anteriormente quando não havia nenhuma menção ao `super();` na classe `Advisor`:

```
public class Employee {
    public Employee() {
        System.debug(' Inicializei Employee');
    }
}

public class Advisor extends Employee {
    public Advisor() {
        super(); // chama o construtor da classe Mãe
        System.debug(' Inicializei Advisor');
    }
}
```

Assim como o `this()` só pode ser chamado dentro de construtores, o `super()` também traz a mesma característica, e também deve ser o primeiro comando a ser executado dentro do construtor. Não pode haver um `super();` e um `this()` dentro do mesmo construtor, é um ou outro.

Enquanto isso, mais demandas surgiram na Bisso Capital Inc.

Assim como *Joseph Climber*, que não desiste nunca, tudo estava indo bem, com a nossa nova implementação do bônus para os Analistas Financeiros, porém a vida é uma caixinha de surpresas! Atender às exigências dos Analistas Financeiros causou uma série de problemas para a diretoria, fazendo com que todos os cargos tivessem uma exceção. Não existia uma regra específica para todos os funcionários.

Com isso, o time de desenvolvimento começou um ciclo de refatoração, sobrescrevendo o comportamento de `getBonus()` em todas as outras classes. Esse processo foi relativamente rápido, porém deixou uma pequena brecha no código: a classe `Employee` ainda sabe como fazer a bonificação. Para sanar esse problema, foi adotada uma medida *deselegante*: forçaram um retorno de `0`, porque o método precisava retornar um `Decimal`.

O código da classe `Employee` ficou assim:

```
public virtual class Employee {

    virtual
    public Decimal getBonus () {
        return 0;
    }

}
```

Problema sanado, a operação voltou aos trilhos e o time seguiu com seu fluxo de desenvolvimento, sempre com muita demanda.

Quando surgiu uma nova demanda de um novo cargo que deveria ser criado, o de *Vice-Presidente Latam*, como o time tinha muita demanda e era uma atividade relativamente simples, passaram-na para um novo integrante, menos experiente e com menos conhecimento do código em questão, que implementou o código a seguir:

```
public virtual class LatamVicePresident extends Employee {

    public List<Director> directors;

}
```

Código implementado! Mas a vida é uma caixinha de surpresas! Como o time de desenvolvimento estava com muita demanda, não fez a devida auditoria no novo código e não observou que não havia testes para o novo *Vice-Presidente Latam*. Adivinha o que aconteceu durante o pagamento do Bônus? O *Vice-Presidente Latam* teve **Zero** de **Bônus**!

Mas assim como *Joseph Climber*, que não desiste nunca, o time voltou para fazer uma refatoração, removendo a causa raiz do erro em questão. Para isso, eles precisaram voltar para as raízes da Orientação a Objetos e realizar uma pergunta simples: *existe um cargo* `Employee` ?

E a resposta é simples: **não**. A classe `Employee` é simplesmente uma base, que fornece um conjunto de características e comportamentos padrões para todo *Funcionário*. Então por que é possível criar uma instância de `Employee` ?

```
Employee employee = new Employee();

employee.salary = 10000;

Decimal employeeBonus = employee.getBonus();

System.assertEquals (0, employeeBonus );
```

Isso é possível porque permitimos! Não informamos que a classe `Employee` deveria ser utilizada somente como base, que não deveria haver uma instância para ela. Para resolver isso, vamos utilizar o modificador `abstract` .

Classes abstratas

Classes abstratas são classes que não podem ser instanciadas. Elas são utilizadas somente como base para que uma **classe filha** seja criada. Isso ocorre pela utilização do modificador `abstract` declarado na classe. Vamos ver o código de `Employee` com essa modificação:

```
public abstract class Employee {

    virtual
    public Decimal getBonus () {
        return 0;
    }

}
```

Toda classe `abstract` é automaticamente ("automagicamente") `virtual` . Não há necessidade de adicionar o modificador `virtual` à classe.

Agora, se você voltar na classe de teste e a recompilar, receberá o erro `Abstract classes cannot be constructed: Employee` (*classes abstratas não podem ser construídas*). Isso porque a classe `Employee` agora não pode mais ser instanciada, ou seja, não pode mais haver `new Employee (...)`.

> **#se-liga-na-info** 😎
>
> **Classes abstratas e concretas**
>
> Temos agora um novo conceito: classes **abstratas**, que não podem ser instanciadas, e classes **concretas**, que podem ser instanciadas.

Mas somente isso não resolve o nosso problema, que é a implementação do método `getBonus()` em `Employee`. Então vamos pensar sobre o método em si, fazendo algumas perguntas:

1. Existe uma implementação padrão de bonificação para todo `Employee`? Não mais, pois todos os cargos têm uma regra específica.
2. Todo `Employee`, *funcionário*, tem bonificação? Sim, todos os funcionários têm bonificação.

Então não podemos ter uma implementação padrão, mas todos os **filhos** de `Employee` devem ter uma bonificação, `getBonus()`. **Que venham os poderes da Orientação a Objetos!**

Métodos abstratos

Vamos utilizar o conceito de **métodos abstratos**, que são métodos que a **classe-mãe abstrata** não sabe como implementar, mas que toda **classe filha concreta** é obrigada a implementar. Eles são **protótipos** de métodos; são métodos sem corpo, que utilizam do modificador `abstract` para indicar que a classe não precisa implementá-lo.

Vamos refatorar a `Employee` para que o método `getBonus()` seja **abstrato**:

```
public abstract class Employee {

    // campos omitidos
    public abstract Decimal getBonus ();
}
```

Note a ausência de chaves `{ ... }` no método. Isso ocorre somente para os **abstratos**, onde não pode haver corpo. Caso esqueça de remover o corpo, o compilador gerará o seguinte erro: `Abstract methods cannot have a body` (*métodos abstratos não podem ter corpo*).

> **#se-liga-na-info** 😎
>
> Métodos **abstratos** devem ser implementados pela primeira classe **concreta** da hierarquia, lembrando que uma classe **concreta** é uma classe não abstrata.

Com essa nova refatoração em `Employee`, a classe `LatamVicePresident` ficará com o seguinte erro de compilação: `Class LatamVicePresident must implement the abstract method: Decimal Employee.getBonus()`, ou seja, *a classe `LatamVicePresident` precisa implementar o método abstrato `Employee.getBonus`*.

```
public virtual class LatamVicePresident extends Employee {

    // campos omitidos
    public Decimal getBonus () {
        return this.salary * 10; // Nuooó!
    }

}
```

#fica-a-dica 🤔

Classes abstratas podem ter métodos concretos e métodos abstratos. Pense sempre que métodos concretos são aqueles que a **classe-mãe** sabe resolver. E os que ela não sabe, mas necessita, delega para as **filhas** através de métodos abstratos.

Agora tudo foi resolvido! O problema anterior foi sanado definitivamente, sem brechas no código. Como eu disse, assim como *Joseph Climber*, que não desiste nunca, sucesso e todos ficaram satisfeitos, inclusive o *VP*. :)

7.3 Polimorfismo

As demandas nunca param

Recentemente, o time de desenvolvimento da *Bisso Capital Inc.* recebeu a demanda de calcular todas as bonificações. Para isso, podemos criar uma classe que tenha essa responsabilidade, criando um método para calcular as bonificações por cargo:

```
public class BonusCalculator {

    Decimal totalBonus;

    public BonusCalculator () {
        this.totalBonus = 0;
    }

    public Decimal calculateBonus ( Advisor advisor ) {
        return this.totalBonus += advisor.getBonus();
```

```
    }

    public Decimal calculateBonus ( Manager manager ) {
        return this.totalBonus += manager.getBonus();
    }

    public Decimal calculateBonus ( Director director ) {
        return this.totalBonus += director.getBonus();
    }

    public Decimal calculateBonus ( Analyst analyst ) {
        return this.totalBonus += analyst.getBonus();
    }

    public Decimal getTotalBonus () {
        return this.totalBonus;
    }

}

@isTest
public class BonusCalculatorTest {

    @isTest
    public static void testTotalOfBonus () {

        BonusCalculator calculator = new BonusCalculator();

        // comentada a inicialização dos funcionários

        calculator.calculateBonus (advisor);
        calculator.calculateBonus (manager);
        calculator.calculateBonus (director);
        calculator.calculateBonus (analyst);

        System.assertEquals (100000, calculator.getTotalBonus());

    }
}
```

Mas será que essa é a melhor implementação para o cálculo de bonificação? Para responder a essa pergunta, vamos voltar ao conceito de **herança** e fazer a seguinte pergunta: `Advisor`, `Manager` e `Analyst` são `Employee`? Sim!

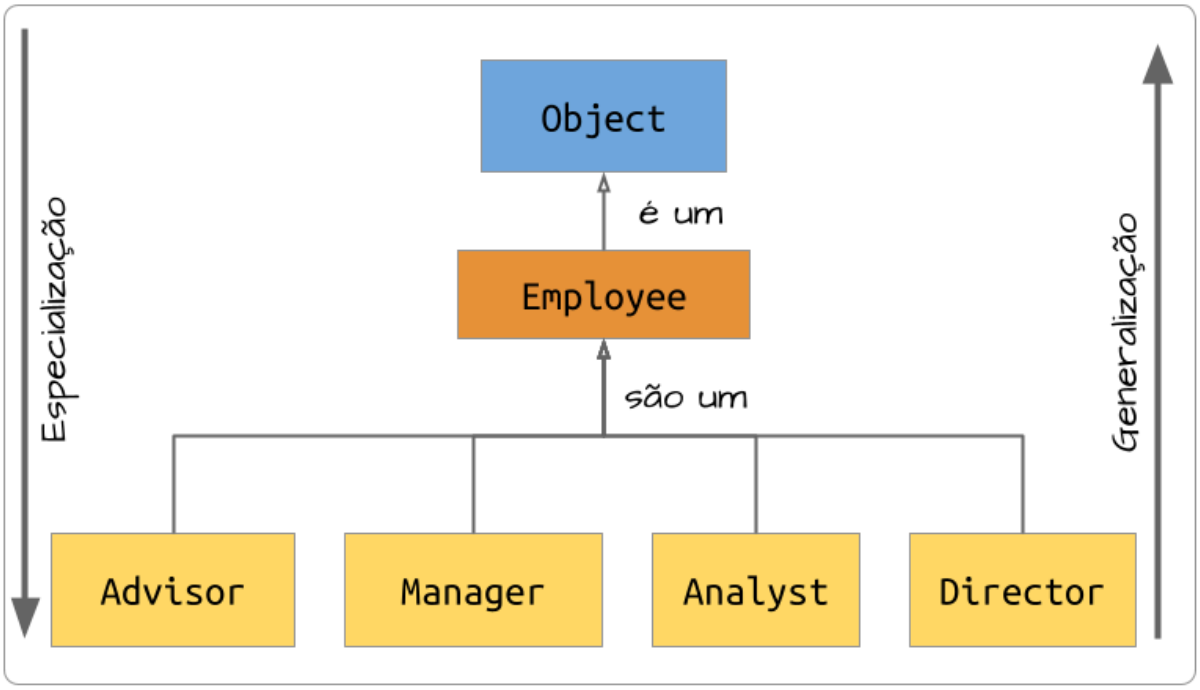


Figura 7.13: Especialização vs. Generalização

Esse diagrama mostra o Princípio da Generalização. Quanto mais ao topo da hierarquia de classes, mais genérica é a classe, enquanto as mais abaixo são as mais específicas.

Com a ideia de generalização e especialização em mente, você pode observar, na imagem a seguir, que, quanto mais genérica for a classe, maior será o número de classes que podem ser essa classe, ou seja, que herdam/estendem essa classe. De forma contrária à generalização, a especialização tem uma única classe acima dela.

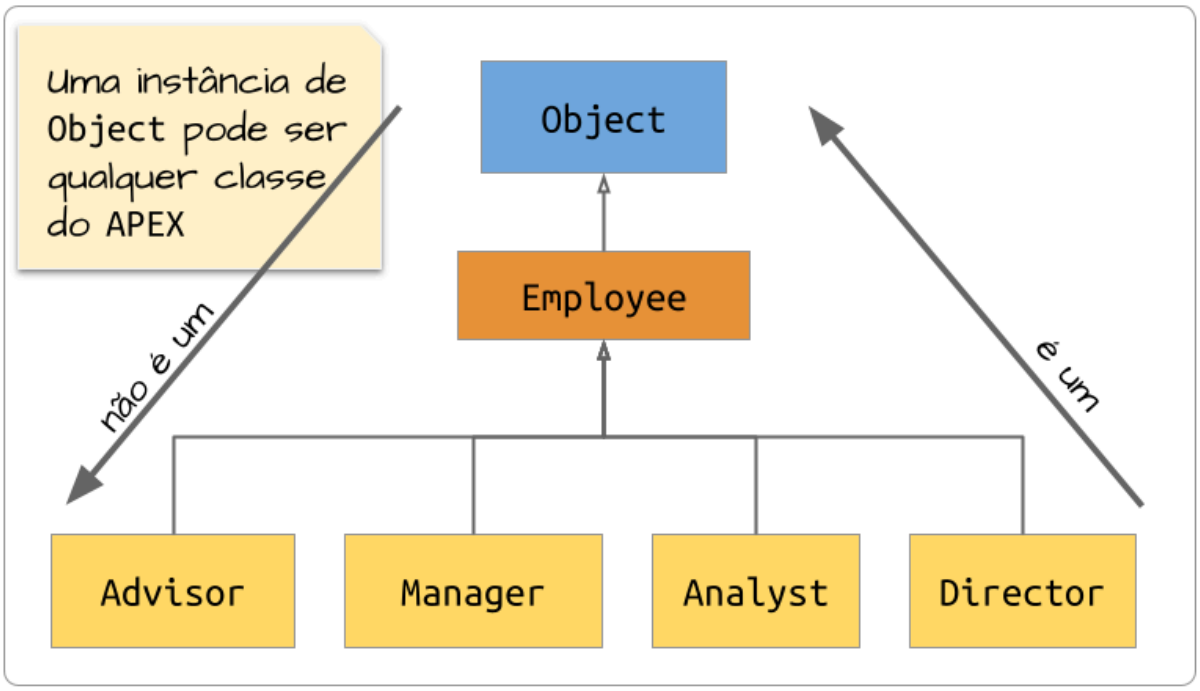


Figura 7.14: Upcasting e downcasting

Então observe o trecho de código a seguir, reavaliando a pergunta "***é um?***":

```
Advisor advisor = new Advisor();

Manager manager = new Manager();

Employee employee = advisor; // Pode, Arnaldo?
```

Aposto que você estava lendo desta forma:

- `advisor` **é um** `Advisor` ?
- `manager` **é um** `Manager` ?
- `employee` **é um** `advisor` ? (Agora "bugou"!)

Vamos alterar a forma como perguntamos:

- A instância de `new Advisor()` é do tipo `Advisor` ? Sim, **é um** `Advisor` ;

- A instância de `new Manager()` é do tipo `Manager`? Sim, **é um** `Manager`;
- A instância de `new Advisor()` é do tipo `Employee`? Sim; se olharmos novamente para a imagem anterior, podemos ver que a classe `Advisor` é filha de `Employee`, então ela **é um** `Employee`.

Esse processo é possível pois ocorre um *Upcast* de forma automática, implícita. Mas tem mais um segredo nessa operação, que contarei logo. Antes desse segredo, preciso que você observe mais um trecho de código e faça a pergunta: ***É um?***

```
Advisor advisor = new Advisor();

Employee employee = advisor; // Pode, Arnaldo?

Manager manager = employee; // Agora bugou!
```

Ih, complicou, não? Vamos ver parte desse código de forma mais gráfica:

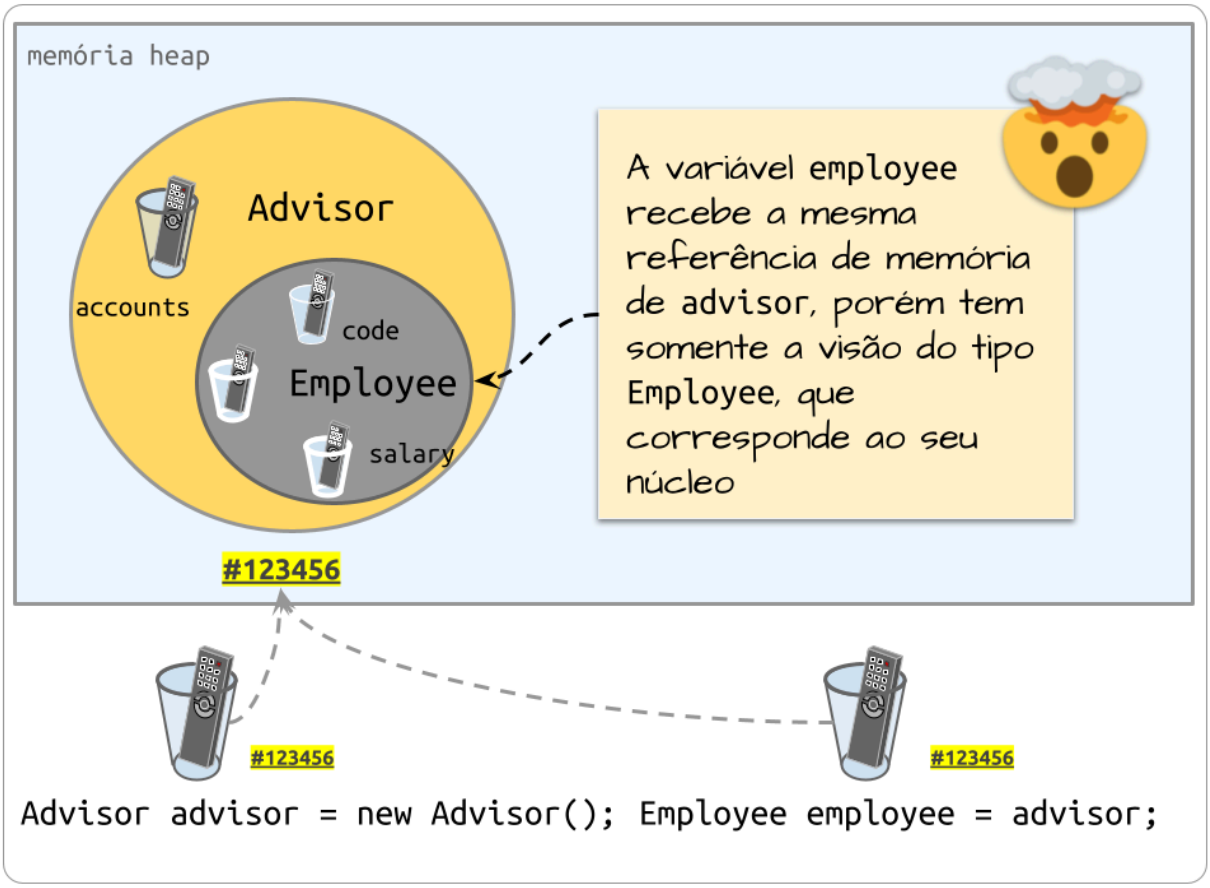


Figura 7.15: Referências de variáveis polimórficas

Vamos avaliar o cenário da atribuição de `Employee employee = advisor;`. Como a instância de `Advisor` possui o núcleo de `Employee`, quando fazemos essa atribuição, a variável `employee` recebe a referência de memória de `advisor`, mas consegue olhar somente para a parte `Employee` de `Advisor`. Isso é o que chamamos de **variável polimórfica**! **SEGREDO REVELADO!**

Usando a analogia do controle remoto, o controle remoto do tipo `Employee` tem a referência de memória de `Advisor`, porém é como se só conseguisse ter acesso às funções de `Employee`. Não temos uma troca de tipo, ele continua forte mesmo com a referência de tipo mais específico.

Agora vamos avaliar a atribuição de `manager = employee;`. Esse cenário é interessante! Perceba que a variável `manager` tenta receber a

mesma referência de memória da variável `employee`, que está apontando para uma instância de `Advisor`. Agora, pergunte-se: `Manager` **é um** `Advisor`? Não!

Somente a resposta a essa pergunta já nos indica que a atribuição de `manager = employee;` não é possível, mas, ao compilar, é gerado o erro `Illegal assignment from Employee to Manager (21:17)`, *Atribuição ilegal de* `Employee` *para* `Manager`. Isso ocorre pelo fato de que há muitos tipos que podem ser `Employee`. Poderia ser um `Advisor`, `Manager` ou `Analyst`. O compilador não consegue fazer de forma implícita essa conversão, ela precisa ocorrer de forma explícita — precisamos fazer um *Downcast*.

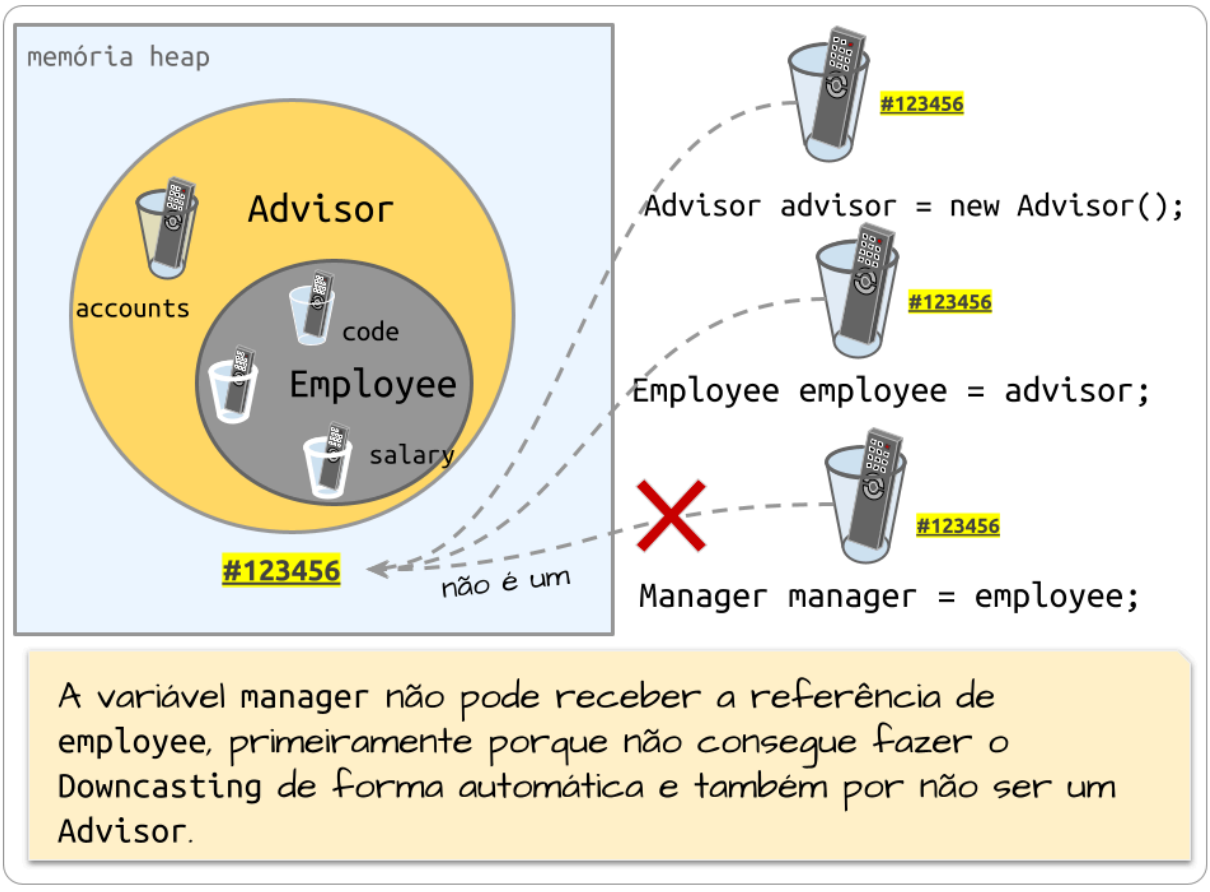


Figura 7.16: Downcast inválido

Vamos ajustar o código para que o compilador reconheça essa atribuição, utilizando o operador de *casting* `(...)`:

```
Advisor advisor = new Advisor();

Employee employee = advisor; // Advisor é um Employee!

Manager manager = (Manager) employee;  // Advisor não
                                       // é um Manager
```

Quando executamos esse bloco de código, recebemos o erro `Invalid conversion from runtime type Advisor to Manager` (*Conversão inválida em tempo de execução, não é possível converter o tipo* `Advisor` *em* `Manager`).

Agora que vimos esses pontos, vamos entrar mais a fundo nos conceitos de **Polimorfismo**!

#agora-o-jogo-vira

Na verdade, você já estava fazendo o uso do **polimorfismo**, sem saber. Quando você redeterminou, sobrescreveu, o comportamento de `getBonus` como em `Advisor` já estava aplicando o conceito.

Então **polimorfismo** permite que objetos de diferentes classes sejam tratados como objetos de uma classe-base comum, mas com comportamentos específicos de suas próprias classes. Ele ocorre quando uma classe filha sobrescreve um método da classe-mãe, permitindo que o comportamento do método seja específico para a classe filha. Já a **variável polimórfica** é a capacidade de uma variável de um tipo mais básico receber a referência de um tipo mais específico.

Vamos voltar para o nosso problema anterior e rever o código implementado para o cálculo da bonificação, ajustando para usufruir desse poder!

```
public class BonusCalculator {
```

```
    Decimal totalBonus;

    public Decimal calculateBonus ( Employee employee ) {
        return this.totalBonus += employee.getBonus();
    }

    public Decimal getTotalBonus () {
        return this.totalBonus;
    }

}
```

Como o código ficou "elegante, pequeno, bonito e cheiroso". Agora ele é absurdamente escalável e flexível, pois pode receber qualquer tipo de funcionário (`Employee`) sem precisar de `ifs` ou vários métodos sobrecarregados (*overloading*).

Podemos executar novamente o nosso teste e verificar que o resultado é o mesmo, mas agora, com o **Polimorfismo**, ele é a aplicação máxima do conceito "**é um**"!

```
BonusCalculator calculator = new BonusCalculator();

// comentada a inicialização dos funcionários

calculator.calculateBonus (advisor);
calculator.calculateBonus (manager);
calculator.calculateBonus (director);
calculator.calculateBonus (analyst);

System.assertEquals (100000, calculator.getTotalBonus());
```

Vantagens do polimorfismo

Flexibilidade: O polimorfismo permite que você escreva código mais flexível e reutilizável. Você pode usar uma única interface ou classe-base para trabalhar com diferentes tipos de objetos.

Extensibilidade: Novas classes podem ser adicionadas ao sistema sem a necessidade de modificar o código existente. Por exemplo, se você adicionar uma nova classe `SalesforceDeveloper` que também herda de `Employee`, o código que usa `Employee` continuará funcionando sem alterações.

Redução de código duplicado: O polimorfismo permite que você evite a duplicação de código, pois você pode usar uma única interface ou classe-base para manipular diferentes tipos de objetos.

Agora que vimos diversos *Jutsus* da Orientação a Objetos, vamos colocá-los em prática para melhorar a nossa base de código. O próximo capítulo será bem intenso e uma aplicação direta dos conceitos apresentados.

Capítulo 8
Aplicando os novos Jutsus

Agora que você já viu diversos *Jutsus* e a importância da Orientação a Objetos, podemos aplicar diversas técnicas para evoluir a nossa base de código. Vamos iniciar a nossa jornada para uma base de código mais *elegante, bonita e cheirosa*.

8.1 Evoluindo as classes Repository

Atualmente, nossas classes do tipo `Repository` estão *replicando* as implementações dos métodos básicos de operações *DML*, como `save` e `remove`. Esse tipo de implementação fere o princípio **DRY** (*Don't Repeat Yourself*), que visa evitar a repetição de código. Como já vimos o poder da **herança**, podemos aplicá-la juntamente ao **polimorfismo** para atacar esse problema, deixando esses métodos mais abstratos, mais genéricos, de forma que consigam atuar com qualquer Objeto Salesforce.

Qual é a **classe-mãe** de todos os Objetos Salesforce? `SObject`! Então basta implementar um método que receba uma `List<SObject>` ou um `SObject` que estaremos sendo genéricos e poderemos trabalhar com qualquer Objeto Salesforce. Observe a implementação a seguir:

```
public abstract class AbstractRepository {

    virtual
    public SObject save ( SObject record ) {

        save ( new List<SObject> { record } );

        return record;
```

```
    }

    virtual
    public List<SObject> save (List<SObject> records) {

        if ( records == null || records.isEmpty()) {
            return records;
        }

        upsert records;
        return records;
    }

    virtual
    public void remove ( SObject record ) {

        remove ( new List<SObject> { record } );

    }

    virtual
    public void remove ( List<SObject> records ) {

        delete records;

    }


}

public class AccountRepository extends AbstractRepository {

    // somente métodos específicos
    // de Account como findByXXX

}


public class OrderRepository extends AbstractRepository {
```

```
    // somente métodos específicos
    // de Order como findByXXX

}
```

Com isso, nosso código ganha muito mais musculatura, pois podemos trabalhar com qualquer Objeto Salesforce e deixamos de ter códigos duplicados.

> **#fica-a-dica** 🤔
>
> Agora temos uma implementação de `AbstractRepository` que, de fato, pode ser utilizada em produção. Você pode evoluí-la, seguindo todos os princípios ensinados até o momento.

Essa nova implementação aplica os conceitos de polimorfismo, herança com classe abstrata e sobrecarga de parâmetros — tudo em uma única implementação! Veja como ela ficou simples, enxuta e elegante. Aplicamos a prática de determinar o comportamento comum e encapsulá-lo. O ponto aqui é sempre voltar à pergunta "***é um?***" para garantir que essa máxima não está sendo quebrada.

8.2 Implementando o padrão TriggerHandler

As *Triggers* desempenham um papel **extremamente importante** em customizações dentro da plataforma Salesforce, porém elas têm uma desvantagem na forma da sua estrutura de código. Seu código é semelhante ao de um único método, tendendo a ficar muito complexo e desorganizado.

Adotamos um padrão, que é manter uma única *Trigger* por objeto, pois assim conseguimos garantir ordem de execução e centralizar comportamentos comuns. Vamos olhar a nossa implementação da `trigger` de `Account` e de `Contract`:

```
/**
 * @author eduardo.bisso
 */
trigger Account on Account ( before insert,
                             before update,
                             after insert  ) {

    List<Account> newAccounts = Trigger.new;
    Map<Id, Account> oldAccounts = Trigger.oldMap;

    AccountValidator validator = new AccountValidator ();
    AccountEventEnricher enricher = new AccountEventEnricher();

    switch on Trigger.operationType {

        when BEFORE_INSERT, BEFORE_UPDATE {

            validator.validate (newAccounts);

        }

        when AFTER_INSERT {

            enricher.scheduleFirstAdvisorMeeting (newAccounts);

        }
    }

}
```

```
/**
 * @author eduardo.bisso
 */
trigger Contract on Contract(after update) {
```

```
    List<Contract> newContracts = Trigger.new;
    Map<Id, Contract> oldContracts = Trigger.oldMap;

    ContractFilter filter = new ContractFilter ();

    InactivateOriginalContractEnricher
            inactivateOriginalContract =
                 new InactivateOriginalContractEnricher ();

    switch on Trigger.operationType {

        when AFTER_UPDATE {


            List<Contract> amendmentContracts =
                    filter.byChangedToSignedStatus (
                                newContracts,oldContracts );

            inactivateOriginalContract
                    .inactivatedBy ( amendmentContracts );


        }

    }

}
```

#fica-dica-clean-code

Gosto de utilizar uma prática do *Clean Code* que é a *regra das três Tabs*, que consiste em você só poder dar três *Tabs* em um método. Tudo que passar da terceira *Tab* deve ser encapsulado em um método/classe. A *Trigger* implementada passa longe dessa prática, parece que estamos desenhando no código :).

Os exemplos apresentados já estão muito melhores do que a nossa primeira implementação, pois tinha tudo junto e misturado: validação, filtro, enriquecimento. Não era possível distinguir o que era o quê!

Agora vamos evoluir o código para deixar de ter código em *Trigger*, utilizando o padrão **TriggerHandler**. Esse padrão resolve os seguintes problemas:

- Descentralização de código de implementação de *Trigger*;
- Tratativa de eventos de *Trigger* de forma mais elegante;
- Remoção de implementações de regras diretamente em *Trigger*.

Para isso, trago uma implementação do Kevin O'Hara que considero uma das melhores atualmente. Você pode obtê-la em: https://github.com/kevinohara80/sfdc-trigger-framework. Essa implementação trata os eventos de forma autoexplicativa, utilizando-se de *sobrescrita* para prover o comportamento de cada evento. A seguir, estão os métodos disponíveis, que cobrem todos os eventos de uma *Trigger*:

- `beforeInsert()`
- `beforeUpdate()`
- `beforeDelete()`
- `afterInsert()`
- `afterUpdate()`
- `afterDelete()`
- `afterUndelete()`

Vamos refatorar o nosso código para começar a utilizar esse novo padrão. Primeiramente, vamos criar a classe `AccountTriggerHandler`, que vai estender a `TriggerHandler`:

```
public with sharing class AccountTriggerHandler
                                    extends TriggerHandler {

    List<Account> newAccounts;
```

```
Map<Id, Account> oldAccounts;

AccountValidator validator;
AccountEventEnricher enricher;

public AccountTriggerHandler () {
    this ( (List<Account>) Trigger.new,
           (Map<Id, Account>) Trigger.oldMap );
}

public AccountTriggerHandler (List<Account> newAccounts
                        ,  Map<Id, Account> oldAccounts) {

    this.newAccounts = newAccounts;
    this.oldAccounts = oldAccounts;

    this.validator = new AccountValidator ();
    this.enricher = new AccountEventEnricher();

}

override
public void beforeInsert () {
    validate();
}

override
public void beforeUpdate() {
    validate();
}

override
public void afterInsert() {
    scheduleFirstAdvisorMeeting();
}

private void validate () {
    this.validator.validate (this.newAccounts);
}
```

```
    private void scheduleFirstAdvisorMeeting () {
        this.enricher
            .scheduleFirstAdvisorMeeting (
                                    this.newAccounts);
    }

}
```

Após a implementação da classe `AccountTriggerHandler`, vamos alterar a *Trigger* adicionando a chamada `new AccountTriggerHandler().run();`. A classe **`TriggerHandler`** utiliza o método `run` para determinar qual evento está sendo executado, direcionando para o método que representa aquele evento. Veja agora como fica a implementação da *Trigger* de `Account`:

```
/**
 * @author eduardo.bisso
 */
trigger Account on Account ( before insert,
                             before update,
                             after insert  ) {

    new AccountTriggerHandler().run(); // LINDO!!!

}
```

Ficou elegante a implementação da nossa *Trigger*! Já a **`TriggerHandler`** parece ter ficado maior — na verdade, ela ganhou mais linhas. Isso porque agora temos construtores e os métodos que processam os eventos. Apesar de ter aumentado o número de linhas, ela melhorou muito a visualização e a capacidade de evolução.

A capacidade de evolução é um ponto a que devemos estar sempre atentos. Para garantir, montei uma listinha de pontos de atenção que devemos observar quando tratamos de *TriggerHandlers*:

- Delegadora de responsabilidades, utilizando-se de composição para realização das tarefas;
- Só deve haver uma *TriggerHandler* por *Objeto Salesforce* (`SObject`);
- Comportamentos esperados em uma *TriggerHandler*: `Filter` , `Validator` , `Enricher` , `Publisher` ;
- Sempre haverá uma implementação de `Filter` ;
- Os métodos sobrescritos — `afterInsert()` , `afterUpdate()` e os outros — devem conter somente chamada para outros métodos da classe, mantendo um único nível de abstração;
- Também utilizamos as *Triggers* para realização de integração com **sistemas externos ao Salesforce** — esse é o papel do `Publisher` , que ainda não foi mencionado aqui.

Veja o diagrama a seguir que ilustra a característica de delegação de uma **TriggerHandler**:

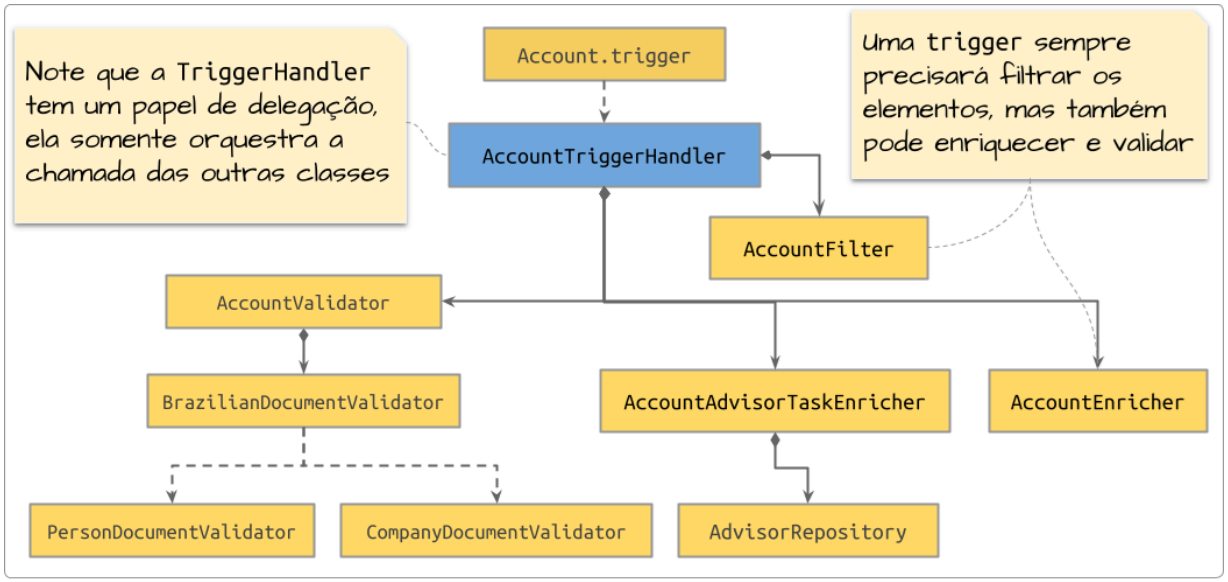


Figura 8.3: Uma TriggerHandler delega para outras classes

8.3 Melhorando nossas classes de teste

No capítulo *Teste tudo!*, você viu como o desenvolvimento orientado a testes é importante para a plataforma, para o projeto e para a saúde do código. Então devemos cuidar dos nossos testes da mesma forma como cuidamos do código "produtivo". Porém, existe uma tendência de que este código seja negligenciado, com abuso de repetição de código, com abandono de convenções de variáveis, tudo porque são classes de teste.

Vamos atacar inicialmente os códigos duplicados de teste, que são apresentados na forma de cenários de teste (aqueles registros que precisamos ter para que um teste execute de forma adequada). Em nosso teste da **Trigger** de `Account`, precisávamos ter registros de `Account` com um CPF ou CNPJ que representasse o cenário *inválido* e outro com cenário *válido*. Isso para garantir que a *Trigger* esteja funcionando corretamente.

Acompanhe os trechos de códigos a seguir que apresentam a cobertura da *Trigger* de *Conta* — `Account.trigger` e `AccountTriggerHandler`:

```
@isTest
static void shouldCreateAccountWithValidDocumentAndCreateTask () {

    //dado (given)
    Account expectedAccount = new Account(Name = 'Teste',
                        DocumentNumber__c = '111.444.777-35');

    // quando (when)
    insert expectedAccount;

    // então (then)
    System.assert (  expectedAccount.Id != null );

    // verificar se a Tarefa foi criada

    List<Task> tasks = [SELECT Id, Subject, Status
                             , WhoId, WhatId
                        FROM Task
```

```
                        WHERE WhatId = :expectedAccount.Id];

    System.assert ( !tasks.isEmpty() ); // retornou resultado

    System.assertEquals ( 1
                        , tasks.size()
                        , 'Apenas uma tarefa deverá ser criada');

}

@isTest
static void
shouldGenerateErrorWhenTryToUpdateAnAccountWithInvalidDocument () {

    //dado (given)
    Account expectedAccount = new Account(Name = 'Teste',
                        DocumentNumber__c = '111.444.777-35');

    insert expectedAccount; // aqui deve gerar um erro

    expectedAccount.DocumentNumber__c = '111.444.777-95';

    try { // <--- tenta executar o bloco de código

        // quando (when)
        update expectedAccount;

        System.assert (false
            , 'Não deveria atualizar a conta com CPF inválido'););

    } catch (DmlException e) { // se o erro for esse
                               // executa o bloco de código

        System.assert (true); // de fato falhou

        // vamos verificar se o erro de
        //fato é de Documento Invalido.
        System.assert (
            e.getDmlMessage(0).contains('Documento Inválido');
```

```
    }

}
```

Note que há blocos muito semelhantes: ambos os métodos apresentam o fluxo de criação de `Account`. Esses blocos de código são chamados de **Fixtures**, que nada mais são do que a preparação de dados para testes. Conforme o apresentado anteriormente, existe uma tendência de duplicação desses cenários, fazendo com o que o código de teste fique muito complexo e difícil de ser entendido, pois, para alguns processos, a dependência de objetos é bem grande.

Para atacar esse problema, apresento o padrão **FixtureFactory**.

FixtureFactory

Vamos segregar a responsabilidade de criação de dados de teste para classes especializadas, chamadas de **FixtureFactory**. Recomendo utilizar uma estrutura de código para essas classes, para que elas fiquem mais legíveis e simples de entender:

1. Classe deve ser anotada como `@isTest` para que possa ser utilizada somente em escopo de Teste, e não ser contabilizada como um código que necessite de cobertura;
2. Métodos chamados de `newNomeObjeto` retornam sempre uma instância de um objeto. Essa instância pode ser utilizada para testes de criação de dados e outros testes em que você não precise de um objeto persistido (salvo) no banco;
3. Observe as regras de obrigatoriedade de campos ao fornecer dados de testes;
4. Métodos chamados de `createNomeObjeto` sempre criam o objeto no banco. Utilizando essa convenção, você consegue identificar, de forma fácil, quando um teste está salvando algo no banco para sua execução;
5. Todos os métodos são *estáticos*, pois não precisam de instância para serem chamados.

O código a seguir apresenta alguns métodos para criação de dados para *Conta* `Account`. Perceba que o método `create` é sobrecarregado para fornecer alternativas de criação do registro e praticamente a mesma assinatura para métodos de criação de instância, `newAccount`:

```
@isTest
public class AccountFixtureFactory {

    public static Account create (String name
                                , String documentNumber) {

        return create ( newAccount (name, documentNumber) );

    }

    public static Account create ( Account account ) {
        insert account;
        return account;
    }

    public static Account newAccount (String name
                                    , String documentNumber) {
        return new Account (Name = name
                , DocumentNumber__c = documentNumber
                , BillingStreet = 'Rua Teste'
                , BillingCity = 'Teste'
                , BillingState = 'SP');
    }

    public static Account fromJson (String payload) {
        return (Account) JSON.deserialize (payload
                                            , Account.class);
    }

    public static List<Account> fromJsonArray (String payload) {
        return (List<Account>) JSON.deserialize (payload
                                        , List<Account>.class);
    }
```

```
}
```

Agora vamos ajustar nossas classes de Teste para utilizar os métodos da `AccountFixtureFactory` :

```
@isTest
static void
        shouldCreateAccountWithValidDocumentAndCreateTask () {

    //dado (given)
    Account expectedAccount =
            AccountFixtureFactory.create ( 'Teste'
                                , '111.444.777-35');

    // então (then)
    System.assert (  expectedAccount.Id != null );

    // verificar se a Tarefa foi criada
    List<Task> tasks = [SELECT Id
                        FROM Task
                        WHERE WhatId = :expectedAccount.Id];

    System.assert ( !tasks.isEmpty() ); // retornou resultado

    System.assertEquals ( 1
                        , tasks.size()
                        , 'Apenas uma tarefa deverá ser criada');

}

@isTest
static void
    shouldGenerateErrorWhenTryToUpdateAnAccountWithInvalidDocument () {

    //dado (given)
    Account expectedAccount = AccountFixtureFactory.create (
                                    'Teste'
                                  , '111.444.777-35');
```

```
        expectedAccount.DocumentNumber__c = '111.444.777-95';

        try { // <--- tenta executar o bloco de código

            // quando (when)
            update expectedAccount;

            System.assert (false
                , 'Não deveria atualizar a conta com CPF inválido');

        } catch (DmlException e) { // se o erro for esse
                                   // executa o bloco de código

            System.assert (true); // de fato falhou

            // vamos verificar se o erro de fato
            // é de Documento Inválido.
            System.assert (
                e.getDmlMessage(0).contains('Documento Inválido');

        }

}
```

Pode parecer que não tivemos muitos ganhos, pois a redução de código na classe apresentada foi bem pequena, porém o ganho não está em uma classe de teste, mas, sim, em um conjunto de classes de testes que poderão utilizar dados de *Conta* para testes.

Agora observe o cenário em que você necessita criar um *Contrato* (`Contract`), que precisa ter *Conta*, *Produto*, *Oportunidade* e *Produto da Oportunidade*. Esse é um cenário bem próximo do real, onde dezenas de *Clientes* utilizam fluxos semelhantes. Como fazer isso de forma mais simples e que possa ser reutilizada? Primeiro, determine a árvore de dependência de Objetos e, em seguida, determine a sequência de criação deles.

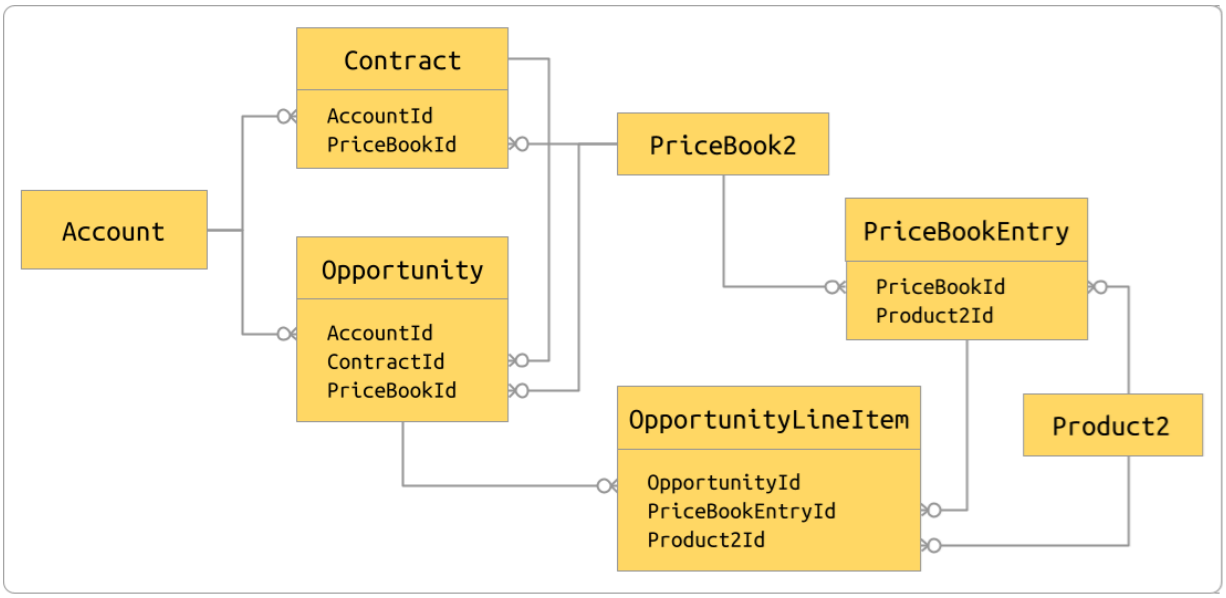


Figura 8.4: Hierarquia de objetos de Contrato e Oportunidade

Conforme a imagem anterior, tanto o objeto `Contract` quanto `Opportunity` dependem de `Account` e `PriceBook2`, porém a sequência pode não parecer tão simples. Então, recomendo que veja se o objeto em questão não tem nenhuma dependência direta, ou se ele só é utilizado a partir de um ponto do processo — como é o caso do `Product2` no exemplo, mas, mesmo assim, sempre precisaremos ter uma sequência de criação. Veja a sugestão a seguir:

1. Criar *Conta* `Account`;
2. Criar um *Produto* `Product2`;
3. Criar um *Produto* do Catálogo `PriceBookEntry`, utilizando o método `Test.getStandardPriceBookId()` para nos prover o Id do `PriceBook2` padrão;
4. Criar a *Oportunidade* `Opportunity`;
5. Criar o *Item* da *Oportunidade* `OpportunityLineItem`;
6. Criar o *Contrato* `Contract`.

Agora vamos ver como fica a dependência de `FixtureFactories` para esse cenário.

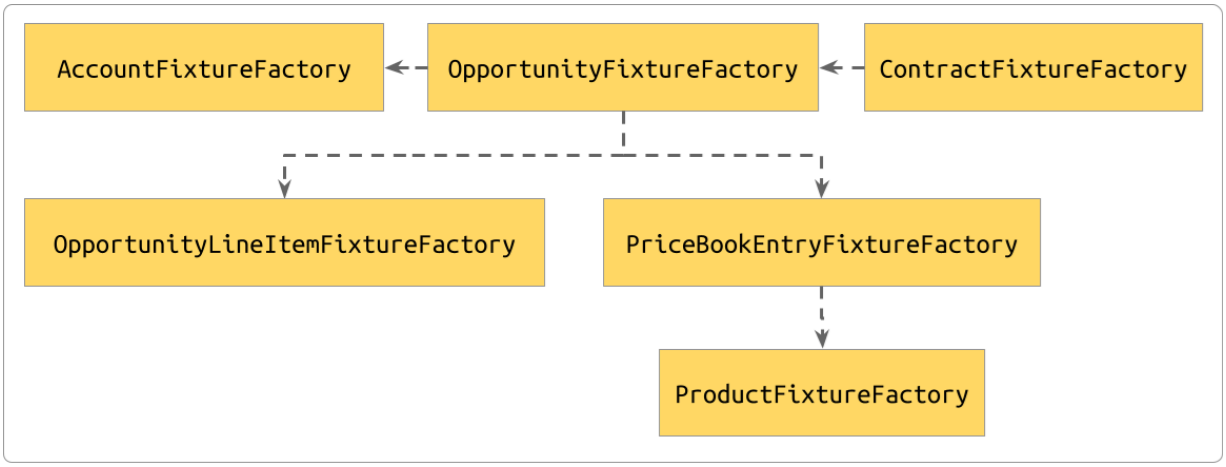


Figura 8.5: Hierarquia das Fixture Factories

Para esse tipo de hierarquia de objetos, gosto de utilizar um padrão que dará a capacidade de aproveitar ainda mais os registros criados anteriormente, sem a necessidade de realizar `SELECTs` para obter os `Ids` criados anteriormente.

```
@isTest
public class ProductFixtureFactory {

    public static Product2 create (String productName) {
        return create ( newProduct(productName) );
    }

    public static Product2 create(Product2 product) {
        insert product;
        return product;
    }

    public static Product2 newProduct(String productName) {
        return new Product2(
            Name = productName,
            Family = 'Teste'
        );
    }

}
```

```
@isTest
public class PriceBookFixtureFactory {

    public static PriceBook2 getStandardPricebook() {
        return new PriceBook2 (
            Id = Test.getStandardPriceBookId(),
            IsActive = true );
    }

}

@isTest
public class PriceBookEntryFixtureFactory {

    public static PriceBookEntry create (PriceBook2 priceBook
                                        , Product2 product ) {

        return create ( newPricebookEntry (priceBook , product));

    }

    public static PriceBookEntry create (
                                PriceBookEntry priceBookEntry) {
        insert priceBookEntry;
        return priceBookEntry;
    }

    public static PriceBookEntry newPricebookEntry(
                PriceBook2 priceBook, Product2 product) {

        return new PriceBookEntry(
            PriceBook2Id = priceBook.Id,
            Product2Id = product.Id,
            UnitPrice = 1.0,
            IsActive = true
        );

    }
```

```
}
```

Antes de continuar com o código da `PricebookEntryFixtureFactory`, observe que dependemos dos dados de *Produto* e *PriceBook*, que também são necessários para criar `OpportunityLineItem`. Para reutilizar esses registros, vamos armazená-los em um `Map<String,SObject>`, dessa forma podemos recuperar, através de uma chave, o registro criado anteriormente.

Esse comportamento será padronizado com o nome de `createIntegratedScenario` para nos referirmos ao fluxo de criação de dados. Observe a implementação a seguir, que está na classe `PricebookEntryFixtureFactory`:

```
public static Map<String,SObject> createIntegratedScenario() {

    Map<String,SObject> records
                          = new Map<String,SObject>();

    Product2 product = ProductFixtureFactory
                               .create('Test Product');

    records.put('product', product);

    PriceBook2 priceBook =
                       PriceBookFixtureFactory
                               .getStandardPricebook();

    records.put('priceBook', priceBook);


    PriceBookEntry priceBookEntry = create (priceBook
                                            , product);

    records.put('priceBookEntry', priceBookEntry);

    return records;
```

```
}
```

Agora observe a implementação de `OpportunityFixtureFactory`, que vai utilizar a implementação de `PricebookEntryFixtureFactory` para criar o `PriceBookEntry` e o `Product2`, que serão utilizados para criar a `OpportunityLineItem`:

```
@isTest
public class OpportunityFixtureFactory {

    public static Opportunity newOpportunity(
                                Account account,
                                PriceBook2 priceBook ) {

        return new Opportunity(
            AccountId = account.Id,
            Name = 'Test Opportunity',
            PriceBook2Id = priceBook.Id,
            CloseDate = Date.today().addDays(1),
            StageName = 'Prospecting'
        );
    }

    public static Opportunity create( Account account,
                                      PriceBook2 priceBook ) {

        return create ( newOpportunity( account, priceBook));
    }

    public static Opportunity create ( Opportunity opportunity ) {
        insert opportunity;
        return opportunity;
    }

    public static Map<String,SObject> createIntegratedScenario() {

        //Aqui é o pulo do gato!
        Map<String,SObject> records =
```

```
                        PricebookEntryFixtureFactory
                                .createIntegratedScenario();

        Account account = AccountFixtureFactory.create (
                                  'Test Account'
                                , '111.444.777-35');

        records.put('account', account);

        PriceBook2 priceBook = (PriceBook2)
                                     records.get('priceBook');

        Opportunity opportunity = create(account, priceBook);

        records.put('opportunity', opportunity);

        Product2 product = (Product2) records.get('product');

        OpportunityLineItem opportunityLineItem =
                        OpportunityLineItemFixtureFactory
                                .create ( opportunity, product );

        records.put('opportunityLineItem', opportunityLineItem);

        return records;

    }

}
```

Dessa forma, fica muito mais simples consumir os dados criados anteriormente. Caso você precise criar todo o fluxo de oportunidade, isso já está pronto, basta chamar o método `createIntegratedScenario()` e obter todos os objetos criados.

#se-liga-na-info 😎

Não crie uma única classe para ser a sua **FixtureFactory**, sempre segregue por **objeto**. Pense sempre no longo prazo, onde ganhamos, e muito, a capacidade de produzir testes de forma muito mais simples e rápida.

8.4 Compartilhamento de registros

Agora que já vimos diversos *Jutsus*, podemos desvendar o último mistério: os modificadores `with sharing`, que utilizamos anteriormente, porém sem nenhum destaque.

Segurança é um dos pilares da plataforma Salesforce e que nos fornece a capacidade de definir regras de visibilidade/permissão a objetos, campos, tipos de registros e mais de uma dezena de outros elementos. Fora todos esses elementos, ela também traz a capacidade de controlar a visibilidade do registro, sendo possível permitir que determinado usuário ou grupo veja determinado registro.

Esse recurso é o que chamamos de **Regra de compartilhamento de registros** (*Sharing Rules*).

Quando salvamos um determinado registro de um objeto, que está configurado como **particular**, a plataforma vai automaticamente definir uma regra básica de compartilhamento para este *Objeto/Registro*: a de que somente o usuário **Proprietário** (*Owner*) pode vê-lo e editá-lo. Essa configuração pode ser alterada para permitir o compartilhamento com base em critérios ou grupos de usuários. Você pode definir grupos de usuários por papéis, por território, entre outras configurações.

Embora as regras de compartilhamento da plataforma sejam muito poderosas, ainda existem cenários em que não é possível realizar o compartilhamento através dela. Um cenário comum é quando estamos utilizando o *Commerce Cloud*, um e-commerce dentro da plataforma. Os usuários que utilizam esse "produto" possuem licenças que podem limitar a visibilidade de registros. Com isso, muitas vezes temos de "driblar" as regras de compartilhamento através de código, e para isso utilizamos os modificadores `with sharing`, `without sharing` e `inherited sharing`. Vamos aos detalhes desses modificadores.

Entendendo o modificador `with sharing`

O modificador `with sharing` declarado em uma classe APEX determina, de forma explícita, que ela executará utilizando as regras de compartilhamento do usuário corrente. Dessa forma, quando a classe for executada, ela terá somente acesso aos registros visíveis pelo usuário que executou a operação.

Por isso, em nossa implementação de *TriggerHandler*, utilizamos esse modificador, para forçar a execução da classe em modo usuário. Veja a seguir, novamente, a declaração da classe aplicando o modificador:

```
public with sharing class AccountTriggerHandler  {

    ContractRepository contractRepository;

    // a classe irá aplicar todas as regras
    // de compartilhamento do usuário corrente

}
```

Agora não temos mais mistérios, mas o que acontece quando omitimos o modificador `with sharing`? Ao omitir esse modificador, estamos indicando para o sistema que esta classe vai executar em modo *default* ou herdar o comportamento da classe mais acima; dessa forma, a classe executará sem aplicar as regras de

compartilhamento ou aplicará a regra determinada pela classe acima.

Em nosso exemplo, a `AccountTriggerHandler` tem uma dependência com a classe `ContractRepository`, que, no caso, não está utilizando o modificador `with sharing`. Nesse caso, ela vai então "herdar" o comportamento da classe `AccountTriggerHandler`, que determinou a execução em modo usuário, `with sharing`:

```
public class ContractRepository  {

    // a classe irá aplicar todas as regras
    // de compartilhamento do usuário corrente

}
```

#se-liga-na-info 😎

Recomendo fortemente esse formato de utilização dos modificadores de visibilidade de registro, onde as classes que estão mais próximas do usuário final, como as `TriggerHandlers`, `Controllers`, `RestResources`. Elas devem determinar o comportamento de execução do modo de compartilhamento (`with sharing`), já que elas têm mais condições de determinar qual é o comportamento desejado. Já as classes mais abaixo sempre serão executadas em modo *default* (sem compartilhamento), ou herdando o comportamento da classe acima.

Mas o que acontece de fato quando executamos a classe no modo `with sharing`?

Imagine o cenário em que temos alguns registros no *objeto* `Contract` criados pelo usuário proprietário, `Sean Forbes`, e outros registros criados por outros usuários, conforme a lista a seguir:

Contract	Account.Name	Account.Owner.Name	Owner
123123	Dickenson	Sean Forbes	Sean Fo
123124	GenePoint	Jack Rogers	Jack Ro
123125	Jackson S.A	Sean Forbes	Sean Fo
123126	Dickenson	Sean Forbes	Jack Ro

Temos a necessidade de contabilizar todos os contratos pendentes, não ativos, dos clientes. Para isso vamos utilizar uma *Trigger* em `Contract` juntamente das classes `ContractTriggerHandler`, `AccountContractEnricher` e `ContractRepository` para coletar os contratos modificados e atualizar os registros de conta. Veja a hierarquia de classes na imagem a seguir:

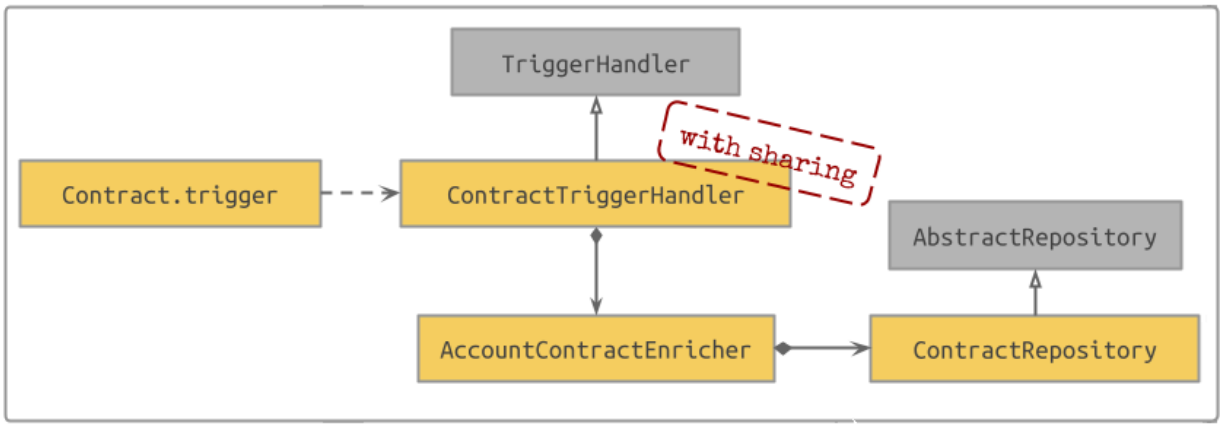


Figura 8.8: Hierarquia de compartilhamento

Como somente a classe `ContractTriggerHandler` está utilizando o modificador `with sharing`, as demais, que não mencionaram o modificador, "herdaram" a regra de compartilhamento da classe mais acima, que, no caso, é a `ContractTriggerHandler`. Com isso, todos os acessos aos registros serão limitados à visibilidade do usuário corrente, que, em nosso caso, é o `Sean Forbes`. Então, quando o

registro com o Contrato `123123` for modificado, ele acionará a *Trigger* atualizando os dados de *Conta*:

```
public with sharing class ContractTriggerHandler
                                    extends TriggerHandler {

    AccountContractEnricher accountEnricher;

    public void afterInsert () {
        enrichWithOpenContracts();
    }

    public void afterUpdate () {
        enrichWithOpenContracts();
    }

    private void enrichWithPendingContracts () {

        accountEnricher.enrichWithTotalPendingContracts(
                                            newContracts );

    }


}

public class AccountContractEnricher {

    ContractRepository repository;

    AccountRepository accountRepository;

    public void enrichWithTotalPendingContracts (
                                List<Contract> contracts ) {

        // converte a lista de Accounts em uma lista de ids
        List<String> accountIds;
```

```
        List<AggregateResult> pendingContracts;

        pendingContracts = repository
                                .findTotalOfPendingContracts(
                                                        accountIds);

        this.enrichAccount( pendingContracts );
        // atualiza o total em account.
    }

    private void enrichAccount (
                    List<AggregateResult> pendingContracts ) {

        List<Account> accounts = new List<Account> ();

        for ( AggregateResult pendingContract : pendingContracts ) {
            accounts.add ( buildAccount ( pendingContract )) ;
        }

        accountRepository.save( accounts );

    }

    private Account buildAccount ( AggregateResult contract ) {
        return new Account (
            Id = (Id) contract.get('AccountId') ,
            TotalPendingContracts__c = (Decimal)
                                        contract.get('total') );
    }


}

public class ContractRepository extends AbstractRepository {

    public List<AggregateResult> findTotalOfPendingContracts (
                            List<String> accountIds ) {

        return [ SELECT COUNT (Id) total, AccountId
```

```
                FROM Contract
                WHERE AccountId IN :accountIds
                  AND Status NOT IN ('Assinado', 'Inativado')
                GROUP BY AccountId];
    }

}
```

O cenário faz com que a classe `ContractRepository` tenha acesso somente aos registros onde o usuário corrente tenha visibilidade. Nesse cenário de contabilizar registros, pode ser que o usuário corrente não tenha acesso a todos os contratos, como o `123126`, cujo proprietário é o `Jack Rogers`. Esse é um cenário bem incomum de execução, pois deveria haver regras de compartilhamento parametrizadas.

Entendendo o modificador `without sharing`

Esse modificador força o sistema a executar sempre ignorando as regras de compartilhamento. Dessa forma, todas as *queries* realizadas obterão todos os registros do banco de dados, independente da visibilidade de compartilhamento. Com isso, você deve utilizar novos elementos na cláusula `WHERE` para obter os registros corretos.

Como comentado anteriormente, um cenário comum que pode acontecer é quando é utilizado o produto *Commerce Cloud*, onde a licença do usuário, por padrão, limita a visibilidade de registros para ler somente os criados pelo próprio usuário. Caso tenha que obter dados de outros objetos do banco de dados, como obter regras de preço especial, por exemplo, temos que utilizar o modificador `without sharing`.

Continuando com o cenário de *Commerce Cloud* em mente, precisamos que todos os *Pedidos* que não utilizem meios de pagamento *Cartão de Crédito* ou *Boleto* sejam enviados para aprovação — dado que, nesse cenário, é necessário avaliar o crédito do cliente. Para isso, a *TriggerHandler* de *Order*, a

`OrderTriggerHandler`, deverá utilizar o modificador `without sharing` para determinar os *Aprovadores de Pedidos* e enviar para aprovação de forma automática:

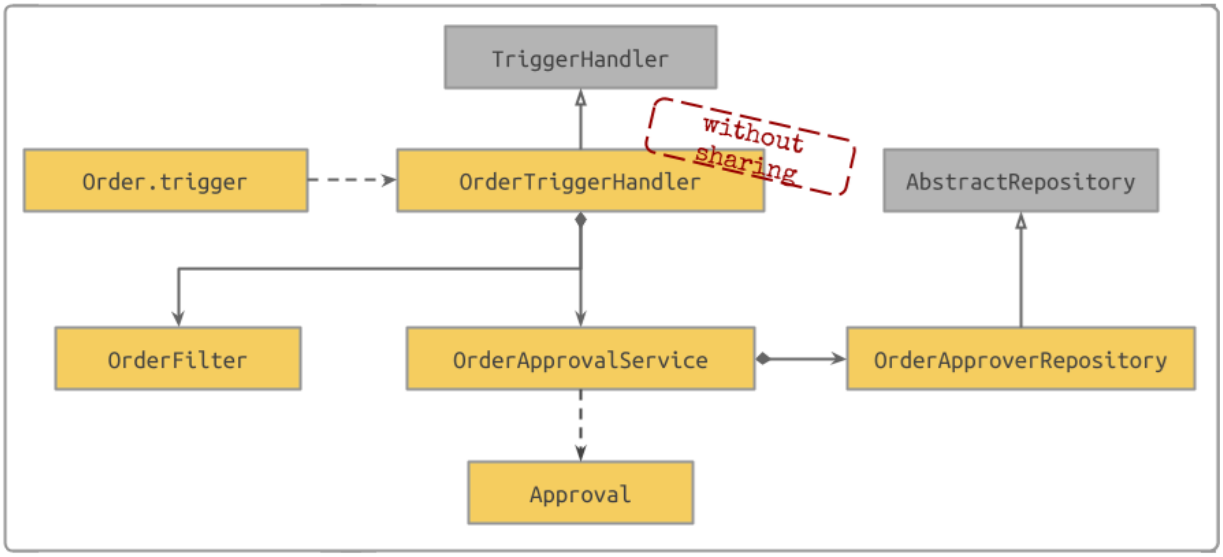


Figura 8.9: Envio para Aprovação em modo sistêmico

```
public without sharing class OrderTriggerHandler
                                    extends TriggerHandler {

    OrderFilter filter;

    OrderApprovalService approvalService;

    OrderApprovalEnricher orderApprovalEnricher;

    // inicialização omitida

    override
    public void beforeInsert () {

        this.enrichWithApprovers();

    }

    override
```

```
    public void afterInsert () {

        this.sendToApproval ();

    }

    private void enrichWithApprovers () {

        List<Order> orders =
                this.filter.byNotCreditCardAndBoleto (
                                                    newOrders);

        orderApprovalEnricher.enrichWithApprovers ( orders );

    }

    private void sendToApproval () {

        List<Order> orders =
                this.filter.byNotCreditCardAndBoleto (
                                                    newOrders);

        approvalService.sendToApproval ( orders );

    }

}


public inherited sharing virtual class OrderApprovalEnricher {

    OrderApproverRepository repository;

    // inicialização omitida

    public void enrichWithApprovers ( List<Order> orders ) {

        List<OrderApprover__c> approvers = findApprovers (
                                                        orders );
```

```
    enrich ( orders, approvers );

}

private List<OrderApprover__c> findApprovers (
                                    List<Order> orders ) {

    // conversão omitida
    List<String> accountIds;

    return repository.findByAccounts ( accountIds );

}

private void enrich ( List<Order> orders
                , List<OrderApprover__c> approvers ) {


    // conversão de approvers para approversByAccount
    // foi omitida
    Map<String, OrderApprover__c> approversByAccount;


    for (Order order : orders ) {

        enrich ( order
            , approversByAccount.get (order.AccountId) );

    }

}

private void enrich ( Order order
                , OrderApprover__c approver ) {

    order.FinancialApprover__c = approver.Financial__c;
```

```
        order.CarrierApprover__c = approver.Carrier__c;

        order.CommercialApprover__c = approver.Commercial__c;

    }

}

public inherited sharing virtual class OrderApprovalService {

    public void sendToApproval ( List<Order> orders ) {

        List<Approval.ProcessSubmitRequest> requests =
                    new List<Approval.ProcessSubmitRequest>();

        for (Order order : orders ) {
            requests.add ( buildRequest (order) );
        }

        Approval.process(requests);
    }

    Approval.ProcessSubmitRequest buildRequest (Order order) {

        Approval.ProcessSubmitRequest approvalRequest =
                            new Approval.ProcessSubmitRequest();

        approvalRequest.setObjectId(order.Id);

        return approvalRequest;

    }

}
```

Esse fluxo vai determinar os aprovadores e enviar para aprovação todos os *Pedidos* (`Order`) diferentes de *Cartão de Crédito* e *Boleto*, ignorando as regras de compartilhamento. Um dos pontos

apresentados foi o modificador `inherited shared`, que propagará o `without sharing` determinado pela classe `OrderTriggerHandler`.

A diferença entre a não utilização do modificador versus a utilização do `inherited shared` é que ele assume, por padrão, o comportamento de `with sharing` quando utilizado em classe com um estereótipo de "Controllers" ou qualquer fluxo que inicie uma transação com banco. Já a ausência do modificador faz com que a classe execute em modo de sistema por padrão, ignorando as regras de compartilhamento de registros.

#se-liga-na-info 😎

Classes com estereótipo de **Controllers** são classes que ficam mais próximas do *front-end*. Esse tipo de classe faz parte do padrão MVC (*Model, View, Controller*), que tem a intenção de segregar a responsabilidade entre a camada de apresentação (*front-end*) e a camada de negócio (*back-end*) da aplicação. São geralmente utilizadas juntamente com os frameworks AURA, Lightning Web Components, Visualforce e na construção de APIs Rest, que não estão no escopo deste livro.

Este capítulo foi longo e intenso em conteúdo, mas necessário para que você consiga implementar códigos literalmente produtivos e elegantes. Aplicando as técnicas e práticas apresentadas aqui, você já conseguirá se destacar no mercado!

Como diria Buzz Lightyear, "*Ao infinito e além!*". Os próximos capítulos serão mais focados em funcionalidades/APIs da plataforma, aumentando o seu kit de ferramentas para produzir *códigos bonitos e cheirosos*!

Parte 3: Trabalhando com serviços remotos

> "*Algumas pessoas gostariam que algo acontecesse. Algumas desejam que aconteça. E outras fazem acontecer.*"
>
> — Michael Jordan

Agora que você já sabe como implementar `Triggers`, `Repositories` e `FixtureFactories` de forma elegante, utilizando vários padrões, vamos dar mais um passo na sua evolução, que é a capacidade de consumir serviços `REST` e expor seu próprios serviços através da **plataforma Salesforce**.

Capítulo 9
Consumindo serviços REST

Um dos maiores benefícios da plataforma Salesforce é a sua conectividade, sua capacidade de expor e consumir APIs com extrema facilidade utilizando o protocolo `HTTP`. Então, para desbravar esses recursos, vamos nivelar o conhecimento do protocolo `HTTP`.

9.1 O que é o HTTP

Para falar de integrações entre aplicações, precisamos conhecer o protocolo `HTTP` (*Hypertext Transfer Protocol*), que é a base para comunicação Web. Ele fornece comunicação entre cliente e servidor através de troca de mensagens, chamada de **requisição e resposta**. A cada requisição, um novo fluxo de dados é criado,

conectando e desconectando. Dessa forma, o servidor consegue atender a mais requisições simultâneas, pois não mantém a conexão ativa nem um estado entre uma requisição e outra.

Essas mensagens têm algumas características que são importantes para o entendimento completo do protocolo e influenciam em nossos serviços.

Request HTTP

A mensagem de `request` (*requisição*) é a responsável por determinar várias informações, como qual o *host*, o *path*, a versão do protocolo, quais são os formatos aceitos pelo "cliente", entre outras. Por exemplo, quando utilizamos o endereço Web `https://viacep.com.br/ws/01001000/json` em nosso *browser*, estamos realizando uma `request` para o serviço da Via Cep. Nessa `request` são passadas várias informações para o servidor, que estão estruturadas de forma simples, conforme mostra a imagem a seguir:

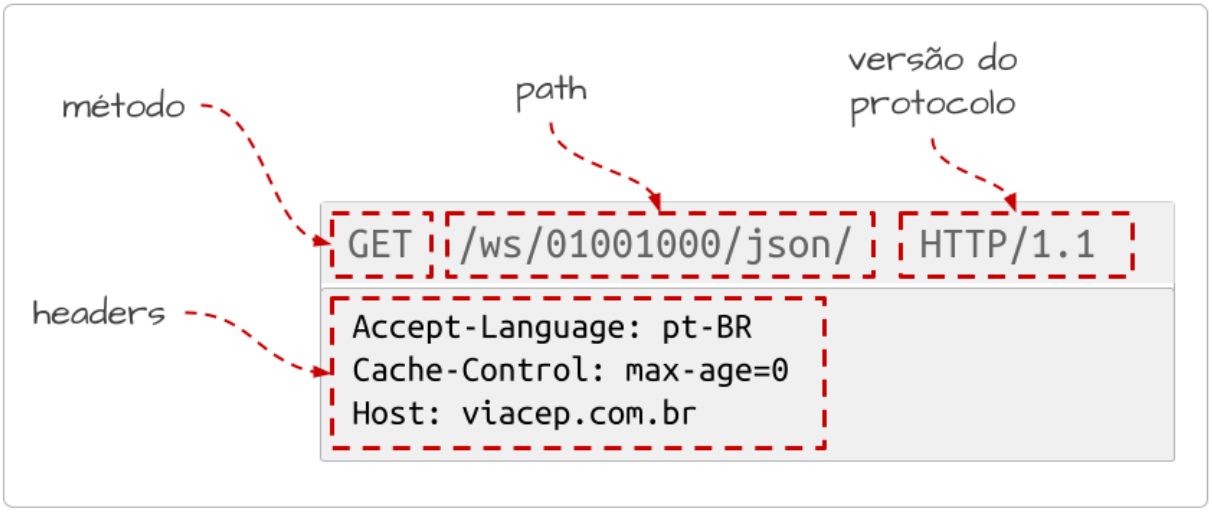


Figura 9.1: Elementos da request

Veja a seguir alguns elementos que são importantes destacar:

- ***Path*:** determina qual a URL que será acessada. No caso do exemplo anterior, estamos realizando uma consulta ao serviço

informando alguns dados, como o `CEP`.

- ***Header*:** é responsável por armazenar várias informações, como `Host`, `Content-Type`, `Accept`, `Authorization`, sendo que seu tamanho é limitado pelo servidor — no caso de NGINX, varia de 4K a 8K;
- **Método:** é responsável por determinar o comportamento do servidor. Através dele, o servidor indica qual operação será realizada. Veja a seguir os métodos mais utilizados:
  - `GET` : utilizado para obter um conteúdo do servidor;
  - `POST` : utilizado para criar um conteúdo no servidor;
  - `PUT` : utilizado para atualizar um conteúdo no servidor;
  - `PATCH` : semelhante ao **PUT**, é também utilizado para atualizar um recurso, porém é usado para atualizar somente parte do recurso;
  - `DELETE` : utilizado para remover um conteúdo do servidor.

Response HTTP

A mensagem de `response` representa o retorno de nossa requisição. Assim como a `request`, ela também possui vários elementos, como demonstrado a seguir:

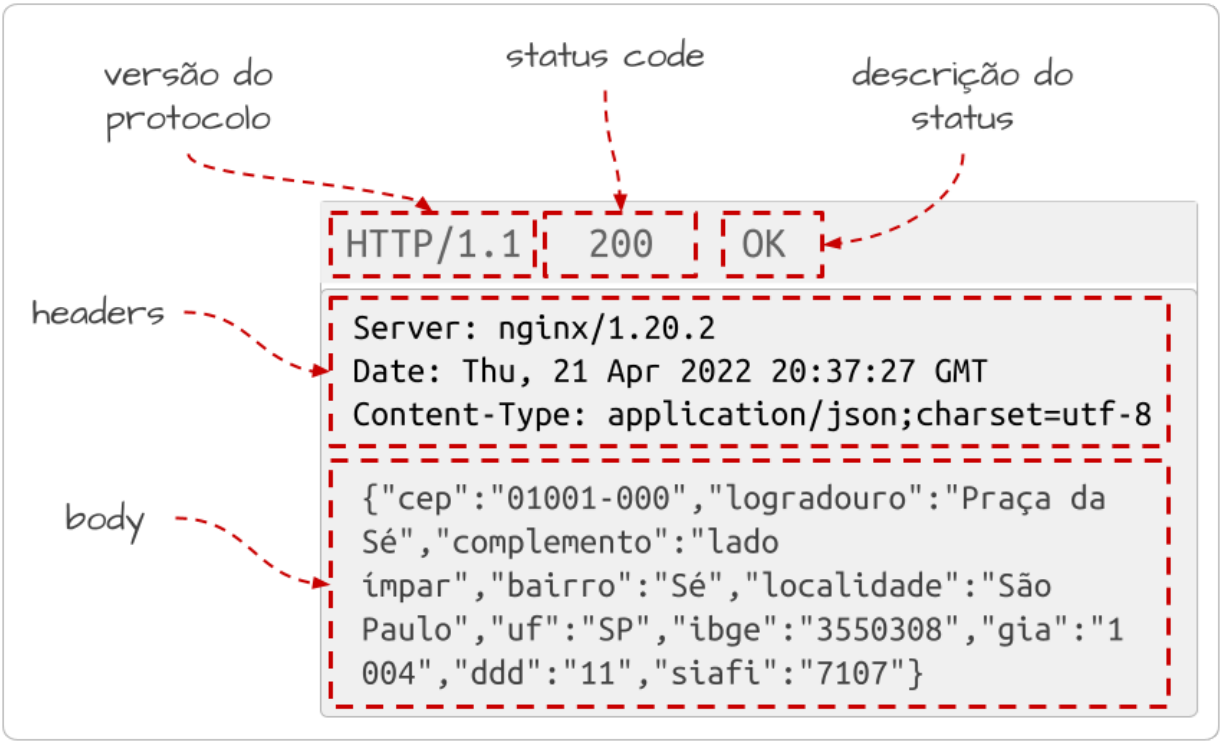


Figura 9.2: Elementos de response

Veja a seguir alguns elementos que são importantes destacar:

- ***Header***: através dos *headers*, podemos identificar diversas informações, tanto do servidor quanto do conteúdo. Veja algumas que podem ser retornadas:
  - `Content-Type` : identifica o tipo do conteúdo retornado, como `text/html; charset=utf-8` , `application/pdf` ;
  - `Server` : identificador do servidor que processou a requisição;
  - `Date` : contém a data e a hora que a mensagem foi gerada.
- **Body:** conteúdo retornado pelo servidor, que pode ser um arquivo, uma página HTML, uma String em JSON, entre outros formatos;
- **Status Code:** determina o status do processamento da *request*. Veja a seguir uma lista de códigos que podemos utilizar para representar o processamento de uma `request` :
  - `100-199` : Respostas de informação;
  - `200-299` : Respostas de sucesso;

- `300-399` : Redirecionamentos;
- `400-499` : Erros do cliente;
- `500-599` : Erros do servidor.

Agora que temos uma visão mais clara sobre o protocolo HTTP, podemos iniciar a construção de consumo de serviços/APIs externas de forma mais clara, entendendo os elementos envolvidos durante a troca de mensagens `request` e `response`.

9.2 Como podemos consumir uma API externa?

Para exemplificar o consumo de uma API externa, vamos utilizar o serviço do Via Cep, com a intenção de obter informações de endereço a partir de um CEP.

> Esse tipo de classe/operação é chamado de *Callout* na comunidade Salesforce.

Observe o trecho de código a seguir, identificando os elementos abordados do protocolo HTTP:

```
HttpRequest request = new HttpRequest();

request.setEndPoint(
            'https://viacep.com.br/ws/01001000/json/');

request.setMethod('GET');

request.setHeader('Content-Type'
                , 'application/json; charset=utf-8');

Http http = new Http();
```

```
HttpResponse response = http.send( request );

if (response.getStatusCode() == 200) {

    System.debug (response.getBody());

}
```

Esse código é relativamente simples: a classe `HttpRequest` é responsável por toda a *configuração* da requisição, informando a URL, o método HTTP e os *headers*. Já a classe `Http` é responsável por *enviar* a requisição, retornando uma instância da classe `HttpResponse`, que é a resposta do serviço propriamente dita. Vários métodos utilizados aqui são autoexplicativos, porém vale ressaltar o método `setHeaders`, que adiciona *headers* à sua requisição. Através dele, podemos informar alguns *headers*, como `Content-Type`, `Accept`, `Authorization`. Através desses *headers* passamos diversas informações para o servidor, como:

- `Content-Type`: tipo do conteúdo enviado;
- `Authorization`: qual o tipo de autenticação que está sendo utilizado, como `Basic`, `Bearer`, entre outras;
- `Accept`: informa quais tipos de conteúdo, expressos como *MIME types*, o cliente é capaz de entender.

Agora que vimos como é simples realizar uma requisição HTTP para um serviço, vamos ver como podemos obter as informações retornadas por ele. O exemplo anterior retorna um conteúdo do tipo JSON, que é uma representação do endereço em um formato de texto, conforme o exemplo a seguir:

```
{
  "cep": "01001-000",
  "logradouro": "Praça da Sé",
  "complemento": "lado ímpar",
  "bairro": "Sé",
  "localidade": "São Paulo",
  "uf": "SP",
```

```
  "ibge": "3550308",
  "gia": "1004",
  "ddd": "11",
  "siafi": "7107"
}
```

Para consumir o retorno do serviço, precisamos criar uma nova chamada, `ViaCepAddressResponse`, que corresponderá à resposta do serviço. Ela terá de seguir, de forma exata, os nomes e tipos dos campos retornados.

```
/**
 * @author eduardo.bisso
 */
public class ViaCepAddressResponse {

    public String cep;
    public String logradouro;
    public String complemento;
    public String bairro;
    public String localidade;
    public String uf;
    public String ibge;
    public String gia;
    public String ddd;
    public String siafi;

}
```

Note que é uma classe simples, com somente os atributos declarados da mesma forma que a API retorna. Com esse tipo de implementação, podemos transformar, de forma simples, o conteúdo retornado em `JSON` em uma instância de uma classe APEX, evitando manipulação de `Strings`.

Para converter o JSON retornado na *response*, utilizamos, em uma instância da classe `ViaCepAddressResponse`, o método `JSON.deserialize()`, que pertence à classe `JSON`. Esse método espera

dois parâmetros: uma `String` e o tipo da classe no qual deverá ser convertida.

```
// preparação da request foi omitida

Http http = new Http();

HttpResponse response = http.send( request );

ViaCepAddressResponse viaCepResponse;

if (response.getStatusCode() == 200) {

    String payload = response.getBody();

    viaCepResponse = (ViaCepAddressResponse)
                        JSON.deserialize(payload
                                , ViaCepAddressResponse.class);

}
```

Essa nova implementação nos faz recordar do capítulo de Orientação a Objetos, pois o método `deserialize` retorna uma instância com a tipagem de `Object`. Por conta disso, precisamos fazer um *downcast* explícito, utilizando `(ViaCepAddressResponse)`, para conseguir atribuir a uma variável do tipo `ViaCepAddressResponse`.

Habilitando a plataforma Salesforce para acessar serviços externos

Segurança é uma das principais características da plataforma, e para que ela possa acessar serviços externos, é necessário realizar uma configuração para habilitar o `host` do serviço externo.

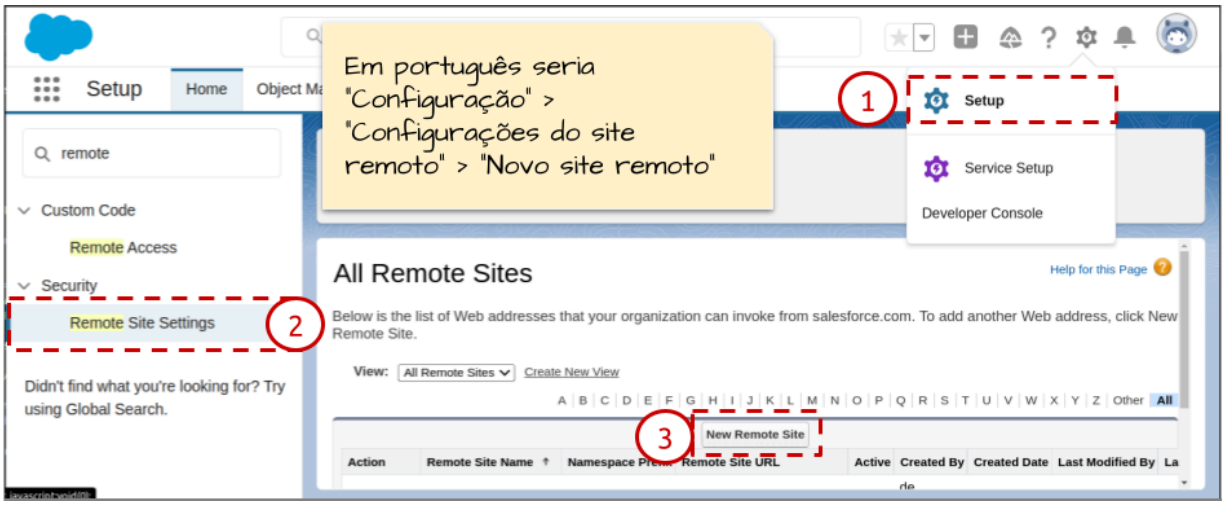


Figura 9.3: Habilitando um site remoto

Em seguida, deve-se informar o apelido para o site remoto e o "domínio". Note que existe uma opção que permite que seja realizada uma `request` não segura. Essa opção é válida quando o serviço não oferece um certificado SSL, ou seja, quando o serviço não é seguro.

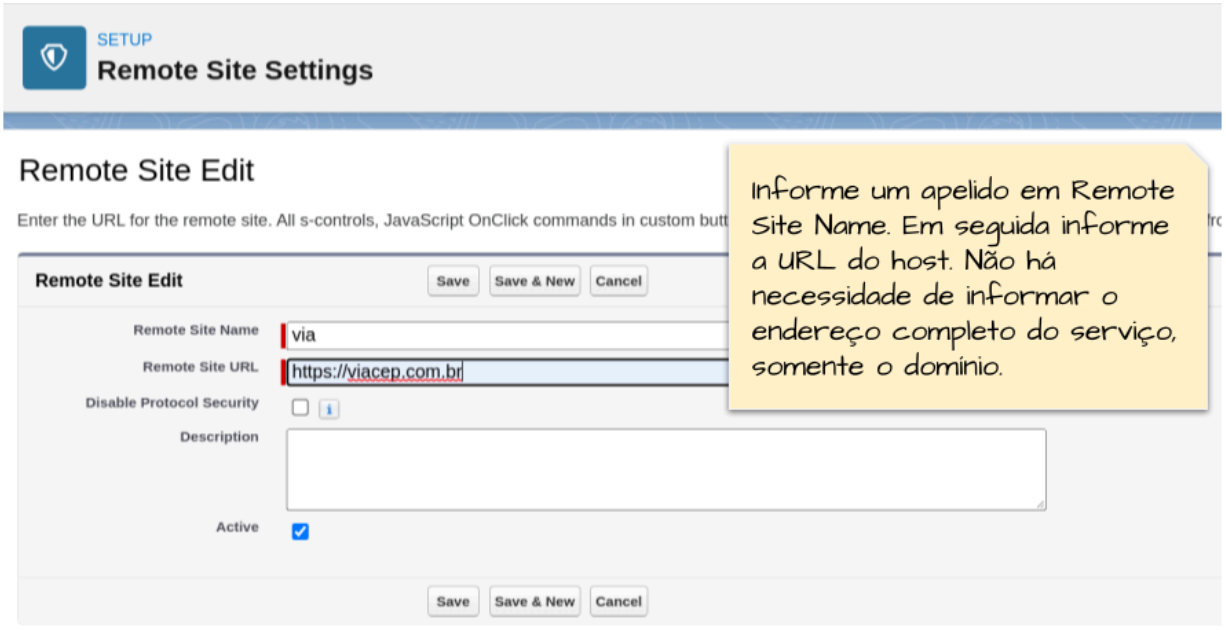


Figura 9.4: Novo site remoto

9.3 Melhorando nossa base de código

Agora que vimos como é simples trabalhar com serviços externos, vamos melhorar nossa base de código para ficar mais elegante e fácil de manter.

Primeiramente vamos criar uma classe que represente o serviço propriamente dito. Ela se chamará `GetAddressOutboundService`. Em seguida, vamos encapsular o código que cria a `HttpRequest` no método chamado `buildRequest`. Também vamos criar outro método para encapsular o tratamento do `HttpResponse` no método `buildZipCodeAddressResponse`.

Podemos expor esse comportamento através do método `findAddressByZipCode`, que retorna uma instância de `ViaCepAddressResponse`:

```
/**
 * @author eduardo.bisso
 */
public class GetAddressOutboundService {

    static final String ZIPCODE_ADDRESS_URL
                                = 'https://viacep.com.br/ws/';

    public ViaCepAddressResponse findAddressByZipCode (
                                            String zipCode) {

        Http http = new Http();

        HttpResponse response = http.send(
                                    buildRequest(zipCode) );

        return buildZipCodeAddressResponse (response);

    }

    HttpRequest buildRequest (String zipCode) {
```

```
        HttpRequest request = new HttpRequest();

        request.setEndpoint( ZIPCODE_ADDRESS_URL
                            + zipCode
                            + '/json/');

        request.setMethod('GET');

        request.setHeader('Content-Type'
                        , 'application/json; charset=utf-8');

        return request;
    }

    ViaCepAddressResponse buildZipCodeAddressResponse (
                                    HttpResponse response ) {

        if (response.getStatusCode() != 200) return null;

        return  (ViaCepAddressResponse)
                    JSON.deserialize ( response.getBody()
                                , ViaCepAddressResponse.class);
    }


}
```

Vamos executar o código no modo anônimo para avaliar a sua execução. Para isso, vamos criar uma instância da classe `GetAddressOutboundService` que será a responsável por realizar a requisição e retornar a resposta:

```
GetAddressOutboundService service
                    = new GetAddressOutboundService();

ViaCepAddressResponse addressResponse;
```

```
addressResponse = service.findAddressByZipCode ('01001000');

System.debug ( addressResponse );
```

A classe `GetAddressOutboundService` ainda poderia ser melhorada? Sim! Vou listar alguns pontos que você pode fazer como desafio:

1. Poderíamos oferecer um mecanismo, parametrizado, para definição de dados como o método `http` , `url` , *headers* etc.;
2. Poderia ter um `RequestBuilder` que seria o responsável por criar a *request* com base na configuração;
3. O fluxo de integração poderia ter somente uma linha ;).

9.4 Testando nossos Callouts

Agora chegou a hora de testar nossa classe `GetAddressOutboundService` , nosso *Callout*. Então vamos criar a classe `GetAddressOutboundServiceTest` , que será a responsável por realizar os testes — lembrando que precisamos cobrir o sucesso e a falha.

Para realizar nossos próximos testes, vamos utilizar uma técnica de escrita de testes chamada de **BDD** (*Behavior-Driven Development*, ou Desenvolvimento Orientado ao Comportamento). O BDD traz uma maior proximidade com a linguagem compreendida pelo usuário final, dado que sempre é focado no comportamento funcional da solução. De forma prática, ele descreve melhor o cenário de teste através de três blocos:

- *Given*: **dado** um determinado contexto, ou seja, uma condição de entrada;
- *When*: **quando** uma ação ocorre no sistema;
- *Then*: **então** é esperado algo.

Dessa forma, nossos testes passarão a utilizar essa notação através do nome do método, ficando `given..When..Then..` . Nós já aplicamos

esse conceito, mas declarado no corpo do método de teste. Essa prática, quando aplicada de forma adequada, nos ajuda a entender o comportamento desejado do método:

```
/**
 * @author eduardo.bisso
 */
@isTest
public class GetAddressOutboundServiceTest {
   // declarado a seguir
}

@isTest
static void
   givenValidZipCodeWhenFindAddressThenReturnCompleteAddress () {

    // implementação do teste.
}
```

Continuando com a nossa implementação, o corpo do método:

```
GetAddressOutboundService service
                    = new GetAddressOutboundService();

ViaCepAddressResponse addressResponse;

addressResponse = service.findAddressByZipCode ('01001000');

System.assert( addressResponse != null
                , 'Esperado um endereço válido');

System.assertEquals ( 'Praça da Sé'
                    , addressResponse.logradouro );
```

Quando executamos esse teste, recebemos um erro `Methods defined as TestMethod do not support Web service callouts` , ou seja, *métodos definidos como TestMethod não suportam Callouts para serviços da Web*.

#e-agora-como-eu-testo 🤷

Para responder a essa pergunta, vamos voltar a um dos princípios de teste de unidade: um teste deve sempre reproduzir o mesmo comportamento, seja de sucesso ou de falha. Ele deve ser consistente. Quando utilizamos serviços externos, não podemos garantir que ele está no ar, que não ocorra nenhum problema. Por conta desse princípio, a Salesforce não permite a execução de *Callouts* em escopo de testes de unidade.

Utilizando `HttpCalloutMock` para testes de serviços externos

Com isso em mente, vou apresentar o conceito de ***Mock***, que é um simulador de comportamento. Em vez de executar a chamada do serviço externo, ele simula essa chamada, retornando um resultado determinado, de sucesso ou falha.

Para implementá-lo, vamos criar uma nova classe, `GetAddressOutboundServiceMock`, que será responsável por simular a resposta do serviço externo. Ela deverá implementar a interface `HttpCalloutMock`, que garante o comportamento de *Mock* (simulação), para chamadas externas. Essa interface determina que o método `respond ( HttpRequest request )` seja implementado. Ele vai retornar uma nova instância de `HttpResponse`, que corresponderá ao resultado esperado pela execução do teste do serviço:

```
@isTest
public class GetAddressOutboundServiceMock
                                    implements HttpCalloutMock {

    String payload;

    public GetAddressOutboundServiceMock (String payload) {
        this.payload = payload;
    }

    public HttpResponse respond ( HttpRequest request ) {
```

```
        HttpResponse response = new HttpResponse ();
        response.setStatusCode(200);
        response.setBody(payload);

        return response;

    }
}
```

Agora vamos utilizar a classe recém-criada `GetAddressOutboundServiceMock` em nosso método de teste, utilizando o método `Test.setMock`:

```
String expectedPayload = '{ "cep": "01001-000", '
                       + ' "logradouro": "Praça da Sé",'
                       + ' "complemento": "lado ímpar",'
                       + ' "bairro": "Sé",'
                       + ' "localidade": "São Paulo",'
                       + ' "uf": "SP",'
                       + ' "ibge": "3550308",'
                       + ' "gia": "1004",'
                       + ' "ddd": "11",'
                       + ' "siafi": "7107"}';

Test.setMock ( HttpCalloutMock.class
         , new GetAddressOutboundServiceMock (expectedPayload) );

GetAddressOutboundService service
                    = new GetAddressOutboundService();

ViaCepAddressResponse addressResponse;

addressResponse = service.findAddressByZipCode ('01001000');

System.assert( addressResponse != null
                , 'Esperado um endereço válido');

System.assertEquals ( 'Praça da Sé'
                    , addressResponse.logradouro );
```

Note que a chamada do método `Test.setMock` recebe dois parâmetros, sendo o primeiro o tipo que será "mockado", e o segundo, a instância da implementação do *Mock*. Isso faz com que a plataforma substitua, em tempo de execução, a chamada interna na classe `Http` pela chamada do *Mock*, o que, graficamente, ficaria semelhante à imagem a seguir:

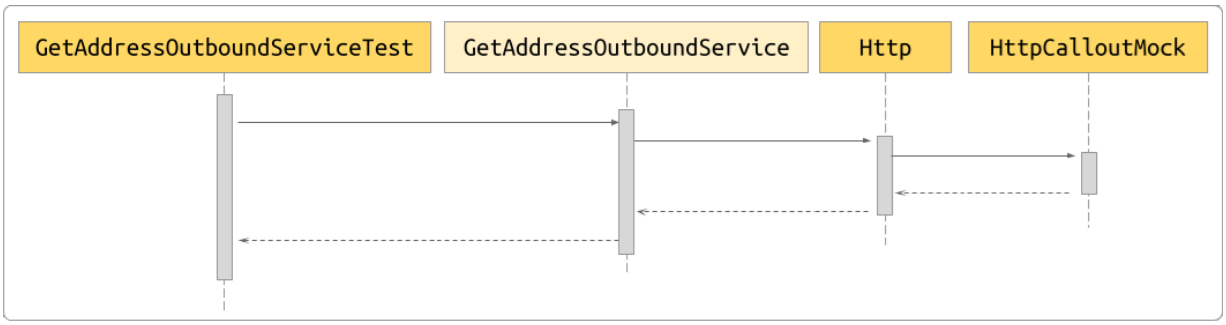


Figura 9.6: Classe mock sendo executada

Como você pode ver na imagem, o método `Test.setMock` substitui parte da execução pela nossa implementação da interface `HttpCalloutMock`, que simula a resposta do serviço externo.

Melhorando classes de teste com FixtureFactory

Agora que vimos como é simples implementar *Mocks* de *Callouts*, podemos melhorar a implementação de nossa classe de teste. Primeiramente, podemos mover a definição do *payload*, nome dado a toda mensagem JSON trocada entre sistemas, para dentro da classe `GetAddressOutboundFixtureFactory`, que será a responsável por prover dados de teste para o serviço `GetAddressOutboundService`.

```
/**
 * @author eduardo.bisso
 */
@isTest
public class GetAddressOutboundFixtureFactory {

    public static String defaultSuccessPayload () {
        return  '{ "cep": "01001-000", '
```

```
                + ' "logradouro": "Praça da Sé",'
                + ' "complemento": "lado ímpar",'
                + ' "bairro": "Sé",'
                + ' "localidade": "São Paulo",'
                + ' "uf": "SP",'
                + ' "ibge": "3550308",'
                + ' "gia": "1004",'
                + ' "ddd": "11",'
                + ' "siafi": "7107"}';
    }

    public static
            ViaCepAddressResponse fromJson (String payload) {

        return (ViaCepAddressResponse)
                    JSON.deserialize( payload
                                , ViaCepAddressResponse.class);
    }

}
```

Vamos ao método de teste utilizando o novo `FixtureFactory`:

```
String expectedPayload = GetAddressOutboundFixtureFactory
                                        .defaultSuccessPayload();

Test.setMock ( HttpCalloutMock.class
         , new GetAddressOutboundServiceMock (expectedPayload) );
```

Já ficou mais elegante! Agora vamos implementar outro método de teste, que visa tratar a falha; porém, para isso, precisamos ajustar a nossa classe `GetAddressOutboundServiceMock` para que possa saber trabalhar com diferentes tipos de código de retorno `Http`. Para isso, vamos definir dois construtores: o *default*, que manterá o comportamento padrão, retornando sempre `200`; e vamos criar um outro que receberá o *payload* e um código de retorno.

```
@isTest
public class GetAddressOutboundServiceMock
                                    implements HttpCalloutMock {
```

```
    String payload;
    Integer statusCode;

    public GetAddressOutboundServiceMock (String payload) {
        this (payload, 200);
    }

    public GetAddressOutboundServiceMock (String payload
                                    , Integer statusCode ) {

        this.payload = payload;
        this.statusCode = statusCode;

    }

    public HttpResponse respond ( HttpRequest request ) {

        HttpResponse response = new HttpResponse ();
        response.setStatusCode (this.statusCode);
        response.setBody (this.payload);

        return response;

    }
}
```

Vamos ao método de teste `givenInvalidZipCodeWhenFindAddressThenReturnNull` (que significa *Dado um CEP inválido, Quando for realizada a pesquisa, Então retornará nulo*), que visa tratar a falha:

```
@isTest
static void givenInvalidZipCodeWhenFindAddressThenReturnNull () {

        // implementação do teste.
}
```

Código do método declarado anteriormente:

```
String expectedPayload = '{}';

Test.setMock ( HttpCalloutMock.class
       , new GetAddressOutboundServiceMock (expectedPayload
                                          , 404) );

GetAddressOutboundService service
                    = new GetAddressOutboundService();

ViaCepAddressResponse addressResponse;

addressResponse = service.findAddressByZipCode ('10010');

System.assert( addressResponse == null
                , 'Esperado um endereço nulo');
```

Tudo coberto! Bonito e cheiroso!

Utilizando *Callouts* na prática

Agora que você já viu como consumir e testar serviços externos, vamos aplicá-lo a um cenário de negócio, que é de fato o que nos motiva a implementar soluções em software. Nosso cenário é realizar o preenchimento de dado de *Endereço* de um `Lead` da empresa *Bisso Capital Inc.* Simples! Vamos ajustar o nosso serviço para fazer isso:

```
public Lead findAddressByZipCode ( String zipCode ) {

    Http http = new Http();

    HttpResponse response = http.send(
                                buildRequest(zipCode) );

    ViaCepAddressResponse viaCepAddressResponse;

    viaCepAddressResponse =
                        buildZipCodeAddressResponse (response);
```

```
    Lead lead = new Lead ();
    lead.Street = viaCepAddressResponse.logradouro;
    lead.City = viaCepAddressResponse.localidade;
    lead.State = viaCepAddressResponse.uf;
    lead.PostalCode = viaCepAddressResponse.cep;

    return lead;

}
```

Qual é o problema dessa implementação? Temos alguns! Primeiro, ela fica fortemente acoplada ao `Lead`; segundo, ela realiza todo o mapeamento de dados direto no método. Então qual seria a implementação ideal? **Segregar responsabilidades** é a resposta:

1. Manter a classe `GetAddressOutboundService` somente com a responsabilidade de realizar a pesquisa e retornar o *Endereço*;
2. Ter uma classe que tem a responsabilidade de obter o **Endereço do Lead**, `GetLeadAddressOutboundService`;
3. Ter uma classe responsável por realizar o mapeamento de dados do `ViaCepAddressResponse` para o Lead, `LeadAddressOutboundTransformer`.

Vamos "codar"! Começando pela `GetLeadAddressOutboundService`, vamos mover o código implementado para nossa nova classe:

```
/**
 * @author eduardo.bisso
 */
public class GetLeadAddressOutboundService {

    GetAddressOutboundService outboundService;

    public GetLeadAddressOutboundService () {
        this.outboundService = new GetAddressOutboundService();
    }

    public Lead findAddressByZipCode ( Lead lead ) {
```

```
        ViaCepAddressResponse addressResponse;

        addressResponse = outboundService
                          .findAddressByZipCode (
                                        lead.PostalCode);

        lead.Street = addressResponse.logradouro;
        lead.City = addressResponse.localidade;
        lead.State = addressResponse.uf;
        lead.PostalCode = addressResponse.cep;

        return lead;
    }

}
```

O código melhorou, mas ainda temos dois problemas. Vamos começar pelo fato de que não estamos mantendo o mesmo **nível de abstração**. Note que o novo método tem um nível de abstração inicial, realizando a chamada do método `findAddressByZipCode`, e passa para outro nível, mais granular, logo em seguida, aumentando a complexidade cognitiva. O segundo problema é de **responsabilidade**. Como comentado anteriormente, ele não tem a responsabilidade de realizar a transformação de dados, mas, sim, de realizar a pesquisa e retornar o `Lead` com os dados do endereço.

#se-liga-na-info 😎

O método apresentado é relativamente pequeno, simples de entender, mesmo com mais de um nível de abstração e excesso de responsabilidade. Porém, quero preparar você para o futuro, onde verá métodos absurdamente maiores e mais difíceis de manter.

9.5 Segregue a responsabilidade com Transformers

Para abordar o segundo problema apresentado anteriormente, vamos conhecer o padrão `Transformer`, que é um padrão de projeto baseado no Pattern Adapter do GOF, porém com uma "roupagem" de integração (veja mais em: https://www.enterpriseintegrationpatterns.com/MessageTransformationIntro.html).

O padrão `Transformer` é o responsável por resolver o problema de mapeamento e transformação de dados, de forma a concentrar a criação e manipulação de dados voltados para integração. Ele também pode ser utilizado juntamente ao *Pattern Builder* com a intenção de prover meios para construção de objetos complexos.

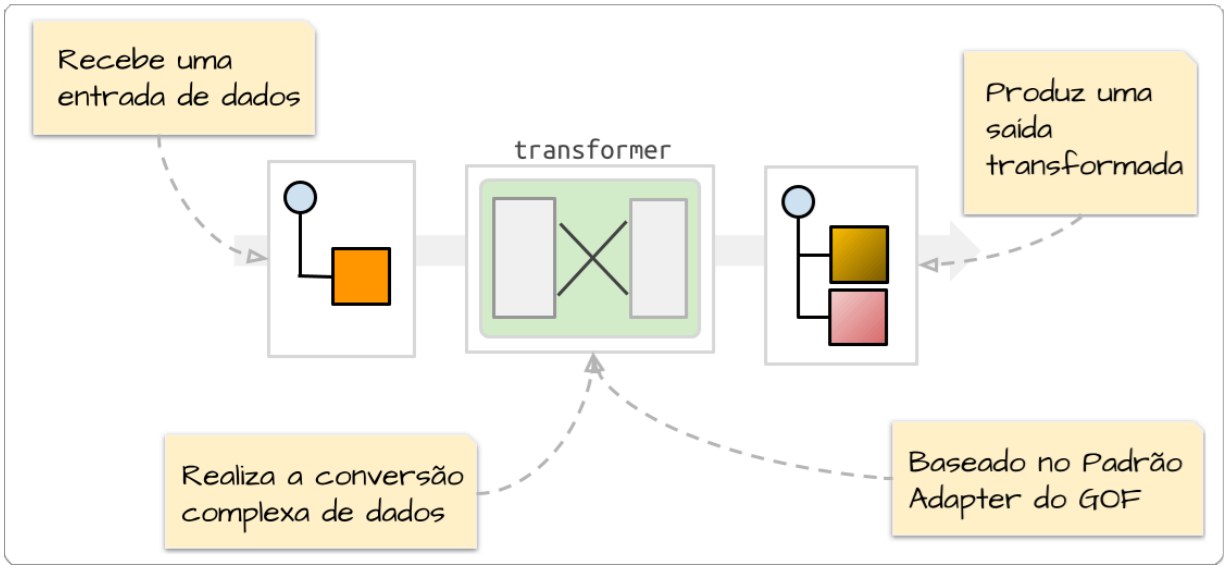


Figura 9.8: Padrão Transformer

Em nosso exemplo, podemos criar uma classe `LeadAddressOutboundTransformer` que receberá a resposta do serviço e realizará a transformação dos dados:

```
/**
 * @author eduardo.bisso
```

```
 */
public class LeadAddressOutboundTransformer {

    public Lead toLead ( ViaCepAddressResponse response
                       , Lead lead ) {

        lead.Street = response.logradouro;
        lead.City = response.localidade;
        lead.State = response.uf;
        lead.PostalCode = response.cep;

        return lead;
    }

}
```

Com isso, a classe `GetLeadAddressOutboundService` pode ser refatorada para utilizar a nossa classe `Transformer` :

```
/**
 * @author eduardo.bisso
 */
public class GetLeadAddressOutboundService {

    GetAddressOutboundService outboundService;
    LeadAddressOutboundTransformer transformer;

    public GetLeadAddressOutboundService () {
        this.outboundService = new GetAddressOutboundService();
        this.transformer = new LeadAddressOutboundTransformer();
    }

    public Lead findAddressByZipCode ( Lead lead ) {

        ViaCepAddressResponse addressResponse;

        addressResponse = outboundService
                          .findAddressByZipCode (
                                              lead.PostalCode );
```

```
        return transformer.toLead (addressResponse, lead);
    }

}
```

Agora observe o diagrama a seguir e veja como a classe `GetLeadAddressOutboundService` delega as responsabilidades para as outras classes. Basicamente, ela desempenha o papel de orquestrar o trabalho, não de realizá-lo!

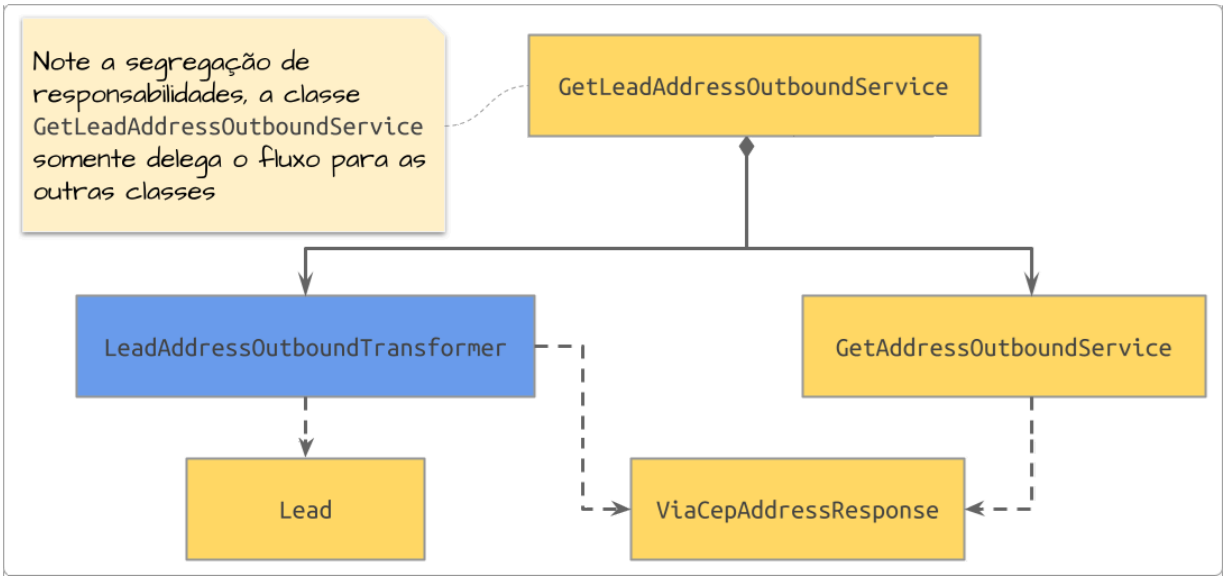


Figura 9.9: Diagrama de classes da GetLeadAddressOutboundService

QUANDO E ONDE DEVO USAR O PADRÃO TRANSFORMER?

O padrão `Transformer` deve ser utilizado em escopos de integração, caso necessite realizar transformação de dados. Em outros escopos, recomendo a nomeação de `Builder` ou `Adapter`, pois o padrão `Transformer` nos remete sempre a integrações, sendo que os demais nos dão contextos mais genéricos.

Testando classes Transformers

O código ficou simples e elegante, agora vamos testar! Vamos começar pela classe `LeadAddressOutboundTransformer`, só que, antes de começar a testar, faça a você mesmo as seguintes perguntas:

1. Qual o objetivo do método `toLead`?
2. Quais são suas entradas e suas saídas?

Provavelmente você respondeu a essas duas perguntas de forma muito semelhante:

1. Transformar a resposta do `ViaCepAddressResponse` em um `Lead`;
2. Entram `Lead` e `ViaCepAddressResponse`, e sai um `Lead` enriquecido com o endereço.

Com isso, podemos organizar a estrutura de dependências de nossa classe de testes. Ela precisará de dados de `Lead` e também de dados de `ViaCepAddressResponse`. Então precisamos de uma classe `FixtureFactory` para `Lead`, `LeadFixtureFactory`, e `GetAddressOutboundFixtureFactory`, pois, como você aprendeu anteriormente, `Fixtures` são dados de teste e `FixtureFactory` são fábricas de dados de teste.

Vamos começar pela `LeadFixtureFactory` e utilizá-la para criar o registro do *Lead*, que será utilizado para testar o método, correto? Não, "bugou"! Este cenário não necessita de um `Lead` persistido, salvo no banco. Ele precisa de uma instância de `Lead` somente, então é exatamente isso que vamos passar:

```
/**
 * @author eduardo.bisso
 */
@isTest
public class LeadFixtureFactory {

    public static Lead newLead (String name
                            , String companyName) {
```

```
        return new Lead (LastName = name
                    , Company = companyName);

    }

}
```

Como a `GetAddressOutboundFixtureFactory` já foi criada, vamos passar para a nossa classe `LeadAddressOutboundTransformerTest` :

```
/**
 * @author eduardo.bisso
 */
@isTest
public class LeadAddressOutboundTransformerTest {
    // métodos serão definidos a seguir
    // por motivos de formatação ;)
}
```

Segue a implementação dos métodos de teste:

```
@isTest
static void givenLeadWhenTransformThenMapAddressToLead() {


    Lead lead = LeadFixtureFactory.newLead('Teste Lead'
                                    , 'Company Teste');

    String payload = GetAddressOutboundFixtureFactory
                                    .defaultSuccessPayload();

    ViaCepAddressResponse response =
                GetAddressOutboundFixtureFactory
                                .fromJson ( payload );

    LeadAddressOutboundTransformer
                    transformer = new
                            LeadAddressOutboundTransformer ();

    Test.startTest();
```

```
        transformer.toLead (response, lead);

        Test.stopTest();

        System.assertEquals ( response.logradouro, lead.Street);
        System.assertEquals ( response.localidade, lead.City);
        System.assertEquals ( response.uf, lead.State );
        System.assertEquals ( response.cep, lead.PostalCode );

}
```

#fica-a-dica 🤔

Um ponto superimportante para classes de teste de `Transformers` é referente às asserções. Recomendo intensamente que você faça **uma asserção por linha de transformação**, pois somente dessa forma você garantirá que o seu código estará funcionando corretamente. Isso porque, em fluxos de integração, a maior incidência de erros ocorre justamente nessa etapa.

Outra dica é manter a classe `Transformer` sem dependências de classes como `Repository` e `Service`, ou seja, classes envolvidas em ciclos de alteração ou pesquisa de dados. Ela recebe um dado e o transforma em outro, e isso faz com que os testes de classes `Transformer` fiquem mais simples, pois não necessitamos ter os dados de fato persistidos no banco de dados.

Continuando com a nossa implementação dos testes, vamos para a `GetLeadAddressOutboundServiceTest`. Como temos uma classe de *Callout* sendo executada, precisamos utilizar a nossa implementação de *Mock* para que ele possa ser executado:

```
/**
 * @author eduardo.bisso
 */
```

```
@isTest
public class GetLeadAddressOutboundServiceTest {
    // implementação a seguir
}
```

O nome do método ficará "grandão", pois visa representar o seguinte cenário de negócio: *Dado um Lead com CEP, Quando ele for pesquisado, Então retornará um Lead com endereço*.

```
@isTest
static void
givenLeadWithPostalCodeWhenFindOutboundAddressThenReturnLeadWithAddress()
{
    // implementação a seguir
}
```

```
Lead lead = LeadFixtureFactory.newLead('Teste Lead'
                                , 'Company Teste');

lead.PostalCode = '01001000';

String payload = GetAddressOutboundFixtureFactory
                                .defaultSuccessPayload();

ViaCepAddressResponse expectedAddress =
                GetAddressOutboundFixtureFactory
                                .fromJson ( payload );


Test.setMock ( HttpCalloutMock.class
       , new GetAddressOutboundServiceMock( payload ));


GetLeadAddressOutboundService service
                    = new GetLeadAddressOutboundService();

Test.startTest();

service.findAddressByZipCode (lead);
```

```
Test.stopTest();

Assert.areEqual ( expectedAddress.logradouro
                    , lead.Street);
```

Resumo

1. Encapsule a responsabilidade de transformar tanto a `request` quanto a `response` para a classe `Transformers` ;
2. Classes de teste de `Transformers` não necessitam de persistir os dados no banco para realização dos testes. Sempre devemos realizar asserções por linha de transformação;
3. Classes de teste que envolvam *Callouts* necessitam de utilização de `HttpCalloutMock` , `Test.setMock(..)` , para realizar o teste;
4. Recomendo a utilização da nomenclatura `OutboundService` para classes que realizam os *Callouts*;
5. Analise a característica do serviço de *Callout*. Caso possa ser reutilizado em escopos de integração distintos, como o nosso serviço de busca de CEP, segregue a responsabilidade em classes distintas;
6. As classes de *Callout* apresentadas são funcionais, factíveis de ir para produção, porém utilize mecanismos de configuração, através de metadados customizados.

Capítulo 10
Expondo serviços REST

Como comentado anteriormente, a plataforma Salesforce fornece diversas opções para integração. Gosto de dizer que ela é *pluggable*, ou plugável, dada a variabilidade de opções de APIs voltadas para integração que podemos utilizar, tais como REST, Bulk, SOAP, Meta Data APIs. Porém, apesar da ampla variedade de opções, muitas vezes necessitamos criar serviços REST, por conta da complexidade de tratamento dos dados, fluxo transacional, rastreabilidade.

Para isso, a plataforma disponibiliza classes do tipo ***RestResource***, que são classes responsáveis por expor serviços REST, com capacidade de processar os métodos HTTP `GET`, `POST`, `PUT`, `DELETE` e `PATCH`.

Classes *RestResource* têm algumas regrinhas importantes:

- Precisa ser global e anotada com `@RestResource`;
- Todos os métodos expostos deverão ser globais e estáticos;
- Suporta somente uma operação de cada tipo — `@HttpPost`, `@HttpGet`, `@HttpPut`, `@HttpPatch`, `@HttpDelete` — por classe.

A plataforma tem implementações distintas para trabalhar com HTTP: no caso de classes do tipo *RestResource*, utilizamos a classe `RestContext` para acessar todos os objetos de contexto atrelados ao *HTTP request* e *HTTP response*.

10.1 Implementando classes RestResource

Para exemplificar os recursos, vamos expor um serviço simples para gestão de *Leads*. Ele vai criar/atualizar e pesquisar um *Lead* através

da classe `LeadController`, que vai expor os métodos de CRUD para o `Lead`:

```
/**
 * @author: eduardo.bisso
 */
@RestResource (urlMapping='/api/lead/*')
global class LeadController {

    @HttpPost
    global static Lead create (Lead lead) {
        LeadRepository repository = new LeadRepository();
        return (Lead) repository.save(lead);

    }

    @HttpPut
    global static Lead store (Lead lead) {
        LeadRepository repository = new LeadRepository();
        return (Lead) repository.save(lead);
    }

    @HttpGet
    global static Lead findById () {

        String id = RestContext.request.params.get('id');

        LeadRepository repository = new LeadRepository();
        return repository.findById(id);

    }

    @HttpDelete
    global static void remove () {

        String id = RestContext.request.params.get('id');

        LeadRepository repository = new LeadRepository();
```

```
        repository.remove( new Lead ( Id = id) );

    }

}
```

Observe que, na declaração da classe, utilizamos uma anotação `@RestResource` informando o parâmetro `urlMapping`, que é o complemento da URL para acesso aos serviços REST.

> A URL de serviços `@RestResource` tem como base `https://<org-instance-name>.my.salesforce.com/services/apexrest`. Dessa forma, os nossos serviços serão sempre essa URL seguida pelo `urlMapping` informado. No caso do nosso exemplo, será `https://<org-instance-name>.my.salesforce.com/services/apexrest/api/lead/`.

Outro ponto importante é o modificador de acesso `global`, que indica que a classe está visível para outras aplicações na plataforma, como no caso de aplicações que estão disponíveis no AppExchange (https://appexchange.salesforce.com).

Vamos avaliar cada um dos métodos implementados na classe `LeadController`, iniciando pelo `create (Lead lead)`. Ele utiliza a anotação `@HttpPost` e recebe um `Lead` como parâmetro, retornando o próprio tipo. Veja novamente como sua implementação é simples:

```
@HttpPost
global static Lead create (Lead lead) {
    LeadRepository repository = new LeadRepository();
    return (Lead) repository.save(lead);
}
```

> Somente **MÉTODOS** anotados com `@HttpPost`, `@HttpPut` e `@HttpPatch` suportam o recebimento de parâmetros; assim, somente eles podem declarar variáveis como parâmetros no método da classe.

A linguagem APEX adota alguns padrões para os métodos que possuem parâmetros, como `create (Lead lead)`. O nome da variável passa a fazer parte do *JSON* que deverá ser enviado para o serviço REST. Observe a declaração a seguir:

```
{
    "lead" : {
        "LastName" : "Carvalho",
        "Company" : "sForce",
    }
}
```

Caso o nome da variável do método seja alterado, o JSON enviado também deverá ser alterado. Por exemplo, se alterarmos para `create (Lead requestedLead)`, o JSON deverá ser alterado para `{"requestedLead" : { ... } }`.

Agora, existe uma maneira de receber um JSON sem a necessidade de informar o nome da variável nele? Sim, basta não informar o parâmetro no método e obtê-lo através da classe `RestRequest`, que é obtida através da `RestContext.request`. Veja o exemplo a seguir:

```
@HttpPost
global static Lead create () {

    RestRequest request = RestContext.request;

    String payload = request.requestBody.toString();

    Lead lead = (Lead) JSON.deserialize(payload, Lead.class);

    LeadRepository repository = new LeadRepository();
    return (Lead) repository.save(lead);
}
```

Note que o código ficou um pouco mais verboso, mas é possível. Então isso será uma escolha, utilizar uma forma ou outra, não há uma recomendação ou uma prática. Só não podemos utilizá-los em conjunto, pois a partir do momento em que declaramos uma variável

como parâmetro do método, a APEX literalmente remove o `requestBody` da `RestRequest`.

Agora que você viu como é simples tratar requisições `POST` e `PUT`, vamos ver como tratar requisições `GET`. Na classe apresentada anteriormente, declaramos o método `findById`, responsável por retornar um *Lead* com base no `id` informado.

```
@HttpGet
global static Lead findById () {

    String id = RestContext.request.params.get('id');

    LeadRepository repository = new LeadRepository();
    return repository.findById(id);

}
```

Como comentado anteriormente, métodos anotados com `@HttpGet` não podem receber variáveis como parâmetro. Para obter os parâmetros da requisição HTTP, devemos utilizar a classe `RestRequest`, que é obtida através da `RestContext.request`. Esses parâmetros são armazenados na forma de um `Map<String, String>` e acessados através do atributo `params` da classe `RestRequest`:

```
    Map<String, String> parameters = RestContext.request.params;

    String id = parameters.get('id');
```

Vale ressaltar que todos os parâmetros recebidos por `GET` são do tipo `String`, então, caso necessite de um tipo diferente, você deverá converter o valor para o tipo desejado.

Os parâmetros que são enviados em uma requisição `GET` são chamados de *Query String*. São aqueles que são informados ao final da URL logo após o sinal de interrogação `?`. Eles devem ser informados como chave e valor, `<nomeParametro>=<valorParametro>`, e separados por um `&`.

Outro método apresentado foi o `DELETE`, que remove um determinado *Lead*. Sua implementação foi simples, porém ele teve uma pequena diferença em relação aos anteriores: o tipo de retorno, que, no caso desse método, foi um retorno `void`, indicando que teremos somente uma resposta de sucesso, retornando um `HTTP CODE 200`. Veja novamente a implementação a seguir:

```
@HttpDelete
global static void remove () {

    String id = RestContext.request.params.get('id');

    LeadRepository repository = new LeadRepository();

    repository.remove( new Lead ( Id = id) );

}
```

#fica-dica-clean-code

De forma geral, classes *RestResource* são bem simples, porém pode haver uma tendência de crescimento. Pense sempre no Princípio da Responsabilidade Única (SRP) e no DRY (*Don't Repeat Yourself*). A classe deve ser uma delegadora e não deve conter operações.

10.2 Como testar classes RestResource?

Só está pronto quando estiver totalmente testado! Como garantimos que estamos retornando uma resposta de HTTP de sucesso, ou mesmo de falha?

As classes do tipo *RestResource* são bem mais simples de testar do que classes de *callout*, pois não precisamos "mockar" a resposta de determinado serviço, só precisamos criar instâncias de `RestRequest` e `RestResponse`, cobrindo os fluxos de nossos métodos.

De forma gradual, vamos passar por todas as etapas de cobertura de nossos testes, começando pelo método de criação do *Lead*, `create (Lead lead)`. Podemos inicialmente tratá-lo como um método comum, que recebe uma instância de um `Lead` e retorna com o `Id` preenchido, indicando sucesso. Veja a implementação a seguir:

```
@isTest
public class LeadControllerTest {
    // implementação a seguir
}

@isTest
static void givenValidLeadWhenCreateThenReturnLeadWithId() {

    // given valid lead
    Lead lead = LeadFixtureFactory.newLead ( 'John'
                                           , 'SForceCompany');

    Test.startTest();

    // when create
    lead = LeadController.create(lead);

    Test.stopTest();

    // then return lead with id
    System.assert(lead.Id != null);
```

```
}
```

Essa implementação é muito semelhante a outras que fizemos anteriormente, mas não estamos cobrindo o fluxo de resposta do HTTP. Isso porque ela está tratando de forma "automágica".

Quando temos métodos que possuem um tipo de retorno específico, a plataforma faz a gestão de *HTTP response* de forma automática, determinando o `Status Code` como 200 para todo método que não gerou um erro. Da mesma forma como vimos para o *HTTP response*, vamos ver como são tratados métodos que necessitam do *HTTP request*. Vamos cobrir o `findById()`, que trata de uma requisição `GET` que utiliza o `RestContext.request.params` para obter os parâmetros.

Para este teste, precisamos criar as instâncias de `RestResponse` e `RestRequest`, sendo que a `RestResponse` é para avaliarmos o `Status Code` e a `RestRequest` é para receber os parâmetros que desejamos utilizar no processamento da requisição. Dessa forma, ao enviar uma `RestRequest` com o parâmetro `id` preenchido, deverá ser retornada uma instância de `Lead` válida:

```
@isTest
static void givenStoredLeadIdWhenFindThenReturnStoredLead () {

    // given stored lead
    Lead expectedLead = LeadFixtureFactory.createLead ( 'John'
                                            , 'SForceCompany');

    RestContext.request = new RestRequest();
    RestContext.request.params.put ('id', expectedLead.Id);

    Test.startTest();

    // when find
    Lead storedLead = LeadController.findById();

    Test.stopTest();
```

```
    // then return stored lead
    System.assertEquals(expectedLead.Id, storedLead.Id);

}
```

O código de teste ganhou mais linhas para tratar a requisição, porém sem prejudicar sua leitura.

10.3 Testando externamente

Agora que temos uma implementação estável testada de forma unitária, vamos testar os nossos serviços externamente através do **Workbench** (https://workbench.developerforce.com), uma aplicação desenvolvida pela comunidade de Salesforce. Essa aplicação traz alguns recursos interessantes, como análise de metadados, realização de *queries*, entre outras funcionalidades, como a capacidade de realizar chamadas REST. Para acessar essa última funcionalidade, navegue no menu `utilities > REST Explorer`.

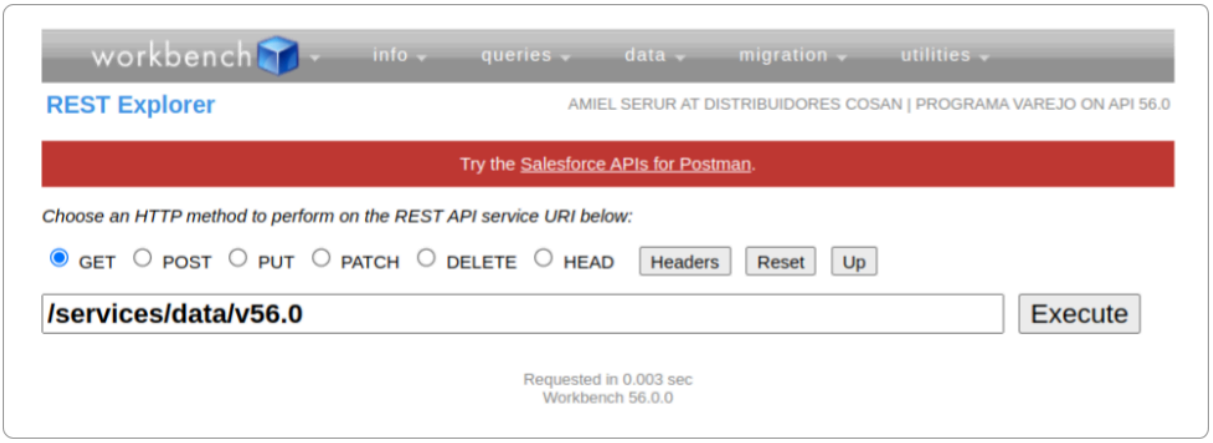


Figura 10.2: REST Explorer

Com essa ferramenta, podemos enviar requisições para nossos serviços REST de forma muito simples, informando o método HTTP, URL e *Headers*, se necessário.

Para testar o serviço de criação de *Leads*, precisamos informar parte da URL, `/services/apexrest/api/lead/` , e o *Request Body* com o JSON de um *Lead* válido, como o apresentado a seguir:

```
{
    "lead" : {
        "LastName": "Bisso",
        "Company": "DevBuilder"
    }
}
```

Com isso, basta pedir para executar, conforme a imagem:

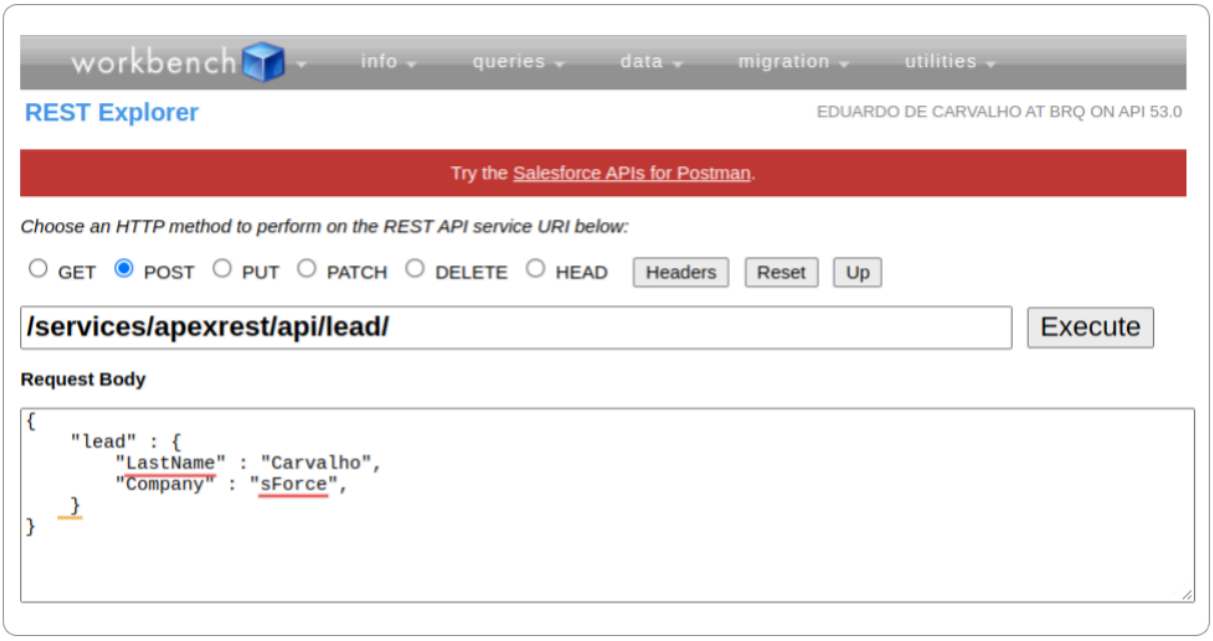


Figura 10.3: Utilizando Workbench para testes

Uma vantagem ao utilizar o Workbench é que não precisamos realizar fluxos de autenticação para execução de nossos serviços; ele se conecta à sua Org de forma "automágica". Então, de forma simples, é possível testar externamente o seu serviço e avaliar o seu comportamento através do protocolo HTTP.

Apesar de ser simples testar pelo Workbench, muitas vezes precisamos enviar para o nosso cliente uma forma mais

documentada de utilizar a nossa API. Para isso, recomendo a utilização da ferramenta **Postman** (https://www.postman.com), que traz diversas funcionalidades, como exemplo de requisição e resposta, pequenas automações, configuração de variáveis de ambiente, entre outras.

Quando utilizamos aplicações como Postman, é necessário realizar o fluxo de autenticação utilizando *OAuth2* (https://help.salesforce.com/s/articleView?id=sf.remoteaccess_oauth_flows.htm&type=5). Ele estabelecerá uma sessão dentro da plataforma, possibilitando a execução de nosso serviço através de um token.

Connected App

Para que uma aplicação externa possa se conectar ao Salesforce utilizando o protocolo OAuth2, devemos utilizar o recurso de *Connected App*, que nos fornecerá as chaves `client_id` e `client_secret`, necessárias para nossa autenticação por meio do Postman.

Veja a seguir um trecho da documentação sobre o *Connected App* da plataforma:

"*Um aplicativo conectado é uma estrutura que habilita um aplicativo externo a integrar-se ao* ***Salesforce*** *usando APIs e protocolos padrão, como SAML, OAuth e OpenID Connect. Aplicativos conectados usam esses protocolos para autenticar, autorizar e fornecer login único (SSO) para aplicativos externos. Os aplicativos externos integrados ao Salesforce podem ser executados na plataforma de sucesso do cliente, em outras plataformas, dispositivos ou assinaturas de SaaS. Por exemplo, quando você faz login em seu aplicativo Salesforce móvel e vê seus dados de sua organização do Salesforce, está usando um aplicativo conectado.*"

Veja mais em Salesforce Help: https://help.salesforce.com/s/articleView?id=sf.connected_app_overview.htm&type=5.

É possível criá-lo utilizando o caminho `Configuração > Aplicativos > Gerenciador de Aplicativos > Novo Connected App` , conforme a imagem a seguir:

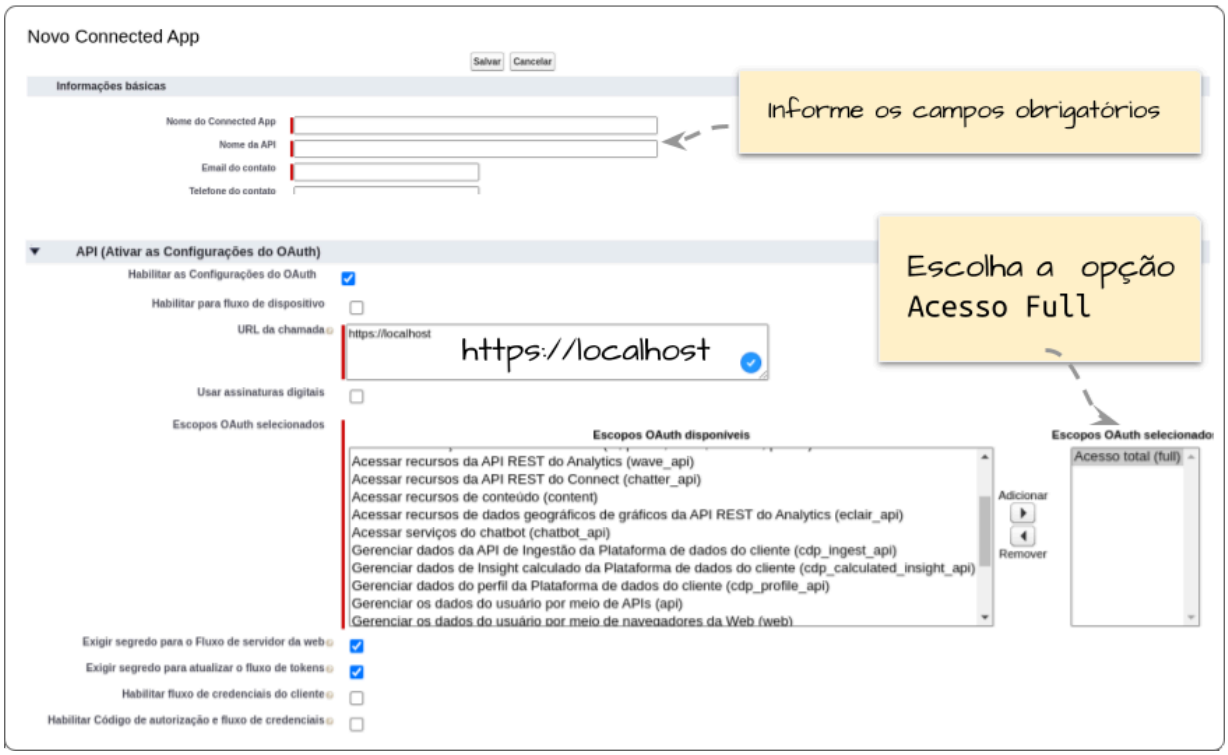


Figura 10.4: Criando uma Connected App

Quando criamos um *Connected App*, obtemos as chaves `client_id` e `client_secret`, que são utilizadas no fluxo de autenticação por OAuth2. Para obtê-las, basta acessar a opção *Gerenciar detalhes do consumidor*, conforme a imagem a seguir:

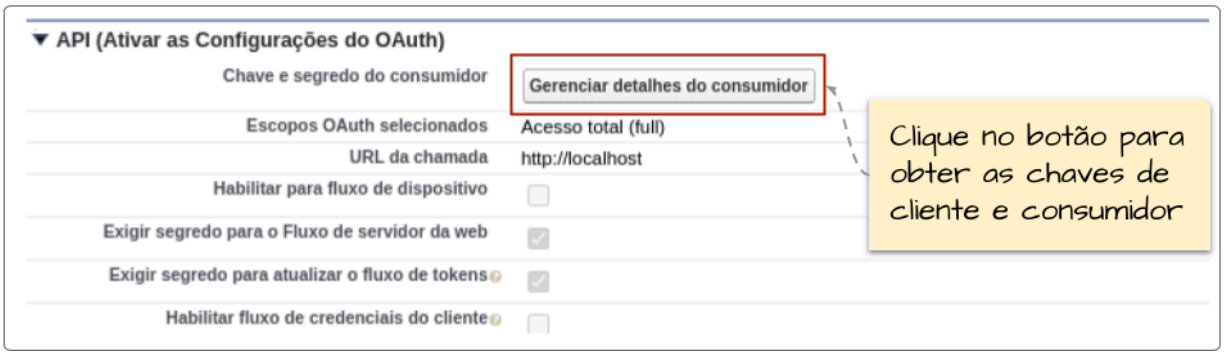


Figura 10.5: Connected App

Após gerar as chaves, será apresentada uma tela semelhante à seguinte, onde você deve copiar os dados de `client_id` e `client_secret`:

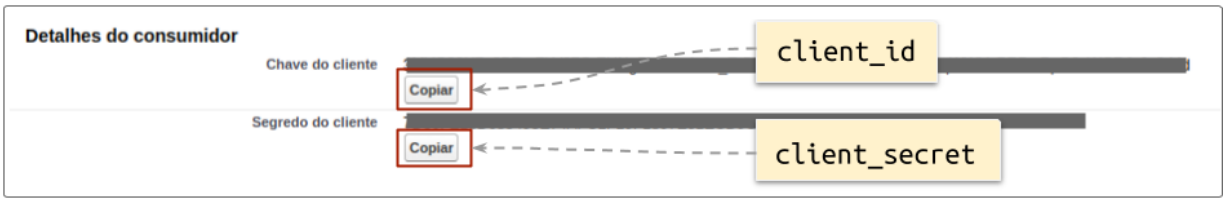


Figura 10.6: Obtendo o client_id e client_secret

Só precisamos fazer mais um ajuste em nossas configurações: precisamos relaxar as políticas de restrição de IPs da Salesforce. Esse passo é necessário porque, por padrão, o Connected App espera que seja configurado um *range* de IPs de acesso. Vamos ajustar para que ele aceite requisições sem a necessidade de realizar esse tipo de configuração:

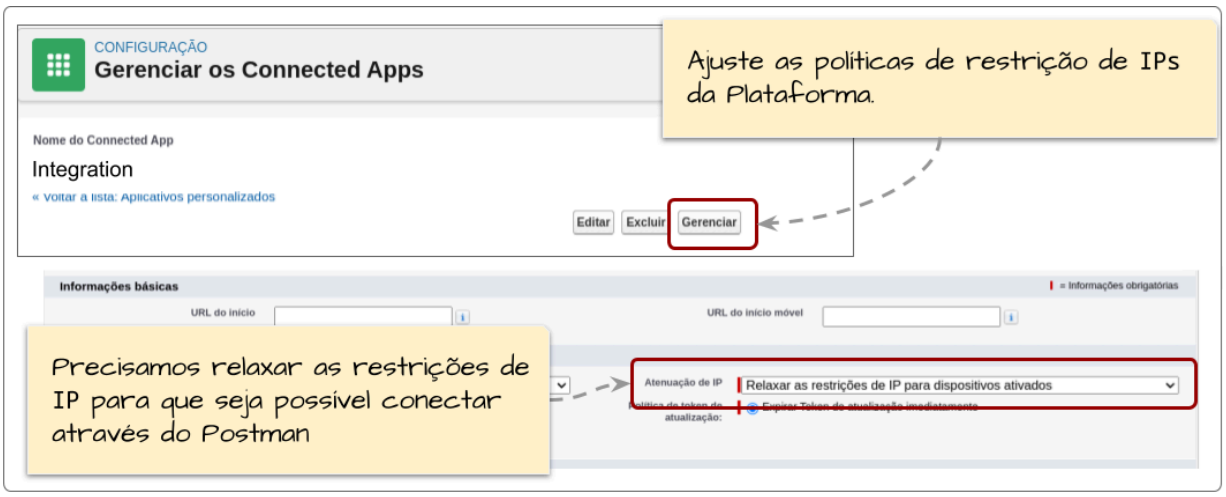


Figura 10.7: Relaxando as políticas de IP

#se-liga-na-info 😎

Em produção, recomendo que estude as políticas de segurança disponíveis no Connected App, para garantir um acesso seguro para o seu cliente.

Com essas informações, podemos realizar a nossa primeira requisição de OAuth2 no Postman. Para isso, vamos criar uma

Collection e, em seguida, uma *Http Request*, conforme a imagem a seguir:

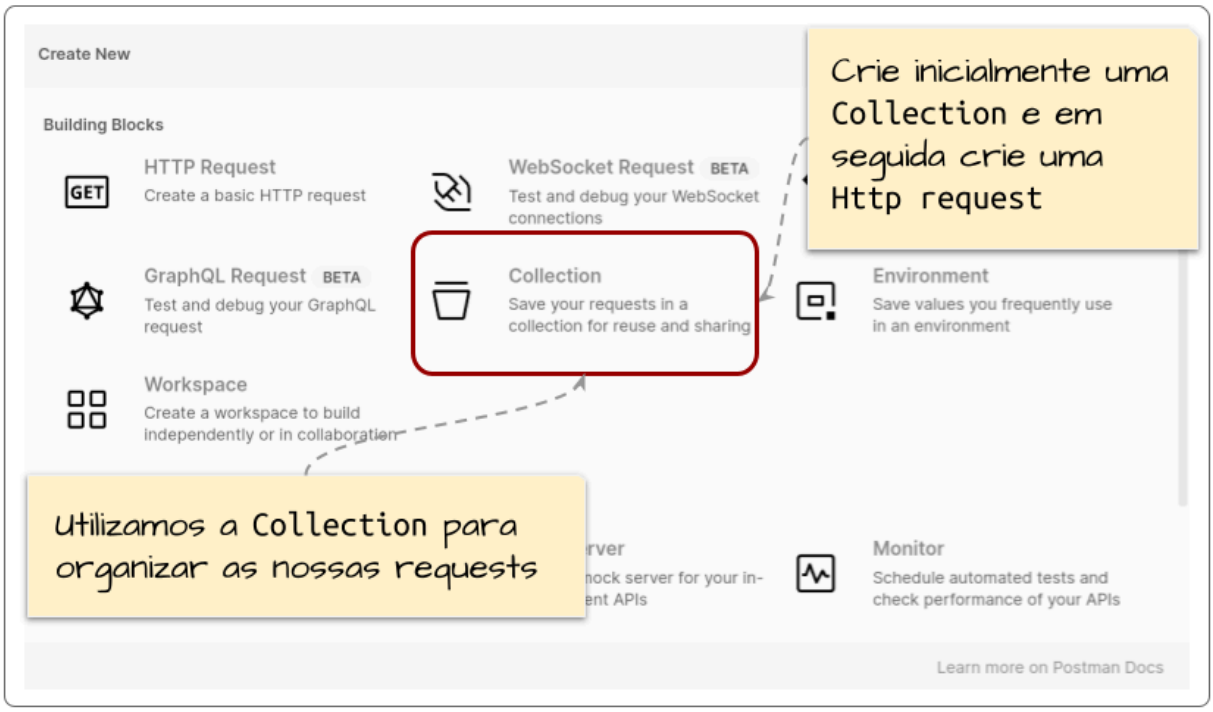


Figura 10.9: Menu New do Postman

A primeira *request* deverá ser a autenticação utilizando o padrão OAuth2. Ela deverá ser criada com os seguintes dados.

Mapeamento do Serviço: OAuth2 Service

- URL: `https://login.salesforce.com/services/oauth2/token`
- Método: `POST`
- *Headers*: `'Content Type' > 'application/x-www-form-urlencoded'`

Mapeamento dos campos da *request*

Nome do campo	Valor
client_id	`[client_id copiado da Org]`

Nome do campo	Valor
client_secret	`[client_secret copiado da Org]`
redirect_uri	`http://localhost`
grant_type	`password`
username	`[seu username]`
password	`[sua senha top secret]`

Veja na imagem a seguir a *request* toda montada:

Figura 10.10: OAuth Request

Após enviar a *request*, você deve receber um *response* semelhante ao apresentado a seguir:

```
{
 "access_token":
"00D4W0000048AiQ!AQoAQKgVN6E4i3uIec7RcHjSHfFcknDSEU6D2QKFybFmMrW566TwI1Yxc
A0aEEeVVsvfAUaUHv5r1PAZzPaFfPQ.XuImOXGt",
 "instance_url": "https://bisso-backendev-dev-ed.my.salesforce.com",
 "id":
"https://login.salesforce.com/id/00D4W0000048AiQUAU/0054W00000B99edQAB",
 "token_type": "Bearer",
 "issued_at": "1601639269587",
```

```
 "signature": "Y4PFBhk+5IhmznVJobqlK6rK8ggSOD41seY4auvYJzA="
}
```

Já estamos aptos a criar a nossa próxima *request* de criação de *Lead*. Para isso, vamos criar uma nova *request* onde utilizaremos parte das informações retornadas pelo *response* do serviço de autenticação. Veja o mapeamento a seguir.

Mapeamento do Serviço: Create Lead

- URL : [instance_url do serviço anterior]/services/apexrest/api/lead
- Método: POST
- *Headers*:
  - Content Type => application/x-www-form-urlencoded
  - Authorization => Bearer [access_token do serviço anterior]

Mapeamento do Body

```
{
   "lead" : {

       "LastName" : "New Lead do Bisso",
       "Company" : "Dev Builder",
       "State" : "SP"
   }
}
```

Após executar a requisição, deverá receber um *response* semelhante ao apresentado a seguir:

```
{
    "attributes": {
        "type": "Lead",
        "url": "/services/data/v5.0/sobjects/Lead/00Q8c0000192laJEAQ"
    },
    "LastName" : "New Lead do Bisso",
    "Company" : "Dev Builder",
    "State" : "SP",
    "Id": "00Q8c0000192laJEAQ"
}
```

Através dessas formas apresentadas, você terá a capacidade de testar plenamente seus serviços, podendo até explorar mais recursos fornecidos pelo Postman, para entregar uma documentação profissional para seu cliente.

Estamos prontos para ir para a próxima fase, onde aplicaremos tudo que vimos em um cenário real.

10.4 Implementando serviços com um cenário mais próximo do real

Chegou a hora de aplicar os conhecimentos em serviços Rest, só que agora vamos ver um cenário de negócio um pouco mais complexo, com fluxo de criação de pedidos através de uma integração. Teremos de trabalhar com vários objetos e seus relacionamentos.

Através desse cenário, poderemos abordar diversos pontos, como exposição de uma API com *request* e *response* customizados, gerenciamento do fluxo de atualização e relacionamento entre os objetos. Observe a imagem a seguir para entender melhor o cenário:

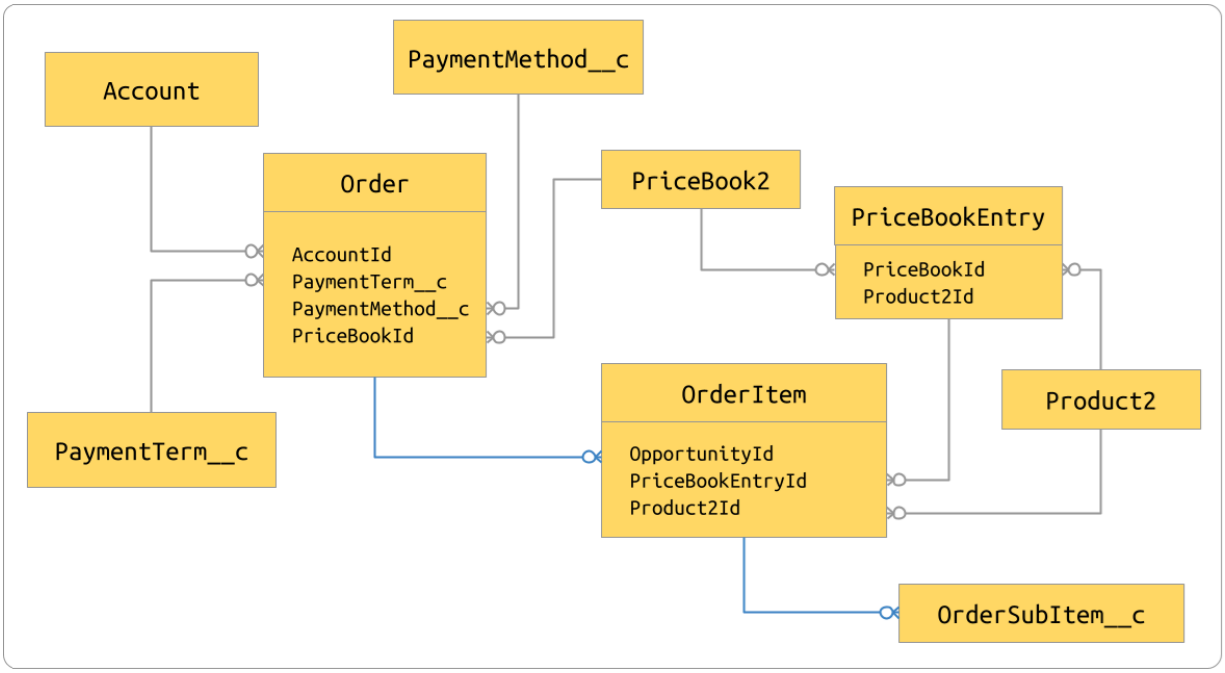


Figura 10.11: Criação de pedido

O diagrama apresenta vários objetos envolvidos nesse processo, alguns são considerados como dados mestres:

- **`Account`** : representa o *Cliente* em questão;
- **`Product2`** : representa o *Produto* que está sendo comprado;
- **`PriceBook2`** : representa o *Catálogo de Vendas*, como catálogo padrão, de preços especiais etc.;
- **`PriceBookEntry`** : determina os produtos de um *Catálogo de Vendas*, como preços especiais etc.;
- **`PaymentTerm__c`** : termo de pagamento, como 30 dias, 60 dias etc.;
- **`PaymentMethod__c`** : determina qual será a forma de pagamento, como cartão de crédito, boleto, entre outras.

Outros objetos são a representação de um *Pedido* que contém itens e subitens em cenários de Pedidos que envolvam kits.

Para implementar esse cenário, em vez de expor os Objetos Salesforce, vamos criar uma classe APEX com a intenção de não acoplar a solução do cliente diretamente aos dados da Salesforce.

Dessa forma, podemos ajustar ou evoluir a estrutura da plataforma sem impactar os consumidores desse serviço. Essa classe será chamada de `PurchaseOrderInboundRequest`.

Geralmente classes com essa característica utilizam *inner classes* em sua construção, com o intuito de encapsular todos os dados relacionados ao objeto. Elas se enquadrariam também no padrão DTO (*Data Transfer Object*), porém não utilizaremos a sufixação `DTO`, mas, sim, a de `InboundRequest` e `InboundResponse`.

```
global class PurchaseOrderInboundRequest {

    public String purchaseOrderNumber;

    public Date orderDate;

    public String accountCode;

    public Address shippingAddress;

    public Address billingAddress;

    public String paymentTerm;

    public String paymentMethod;

    public List<PurchaseItem> items;

    public class PurchaseItem {

        public Integer itemNumber;
        public String productCode;
        public Decimal quantity;
        public Decimal unitPrice;
        public String unitOfMeasure;

        public List<PurchaseSubItem> subItems;

        public PurchaseItem () {
```

```
            this.subItems = new List<PurchaseSubItem>();
        }

    }

    public class PurchaseSubItem {

        public String productCode;
        public Decimal quantity;
        public Decimal unitPrice;

    }

    public class Address {
        public String street;
        public String streetNumber;
        public String streetComplement;
        public String city;
        public String state;
        public String postalCode;
    }
}
```

> Note que a classe `PurchaseOrderInboundRequest` também está utilizando o modificador `global`, isso porque ela também é exposta pela classe `@RestResource`.

Para o *response*, vamos retornar uma estrutura bem simples, que corresponde ao `id` do *Pedido* gerado:

```
global class PurchaseOrderInboundResponse {

    public String id;

}
```

Continuando com nossa implementação, vamos criar a `OrderController`, que vai expor o serviço REST para sincronização de

pedidos:

```
/**
 * @author: eduardo.bisso
 */
@RestResource (urlMapping='/api/orders/*')
global class OrderController {

// métodos implementados a seguir

}
```

Ela vai expor um método de `create` que receberá a `PurchaseOrderInboundRequest` como parâmetro, retornando um `PurchaseOrderInboundResponse` :

```
@HttpPost
global static PurchaseOrderInboundResponse create (
                    PurchaseOrderInboundRequest purchaseOrder ) {
    //

}
```

Agora precisamos mapear os dados da nossa `PurchaseOrderInboundRequest` para os Objetos Salesforce. Para isso vamos utilizar o mesmo padrão adotado em integrações *Outbound* de consumo de APIs externas, o padrão `Transformer` , que será o responsável por mapear os dados de um objeto para outro:

```
public class PurchaseOrderInboundTransformer {

    public Order toOrder (
                    PurchaseOrderInboundRequest purchaseOrder ) {

        // implementação do código virá a seguir
    }

}
```

```
Order order = new Order();

// Será que funciona?
order.AccountId = purchaseOrder.accountCode; //???
order.ErpCode__c = purchaseOrder.purchaseOrderNumber;
order.PoNumber = purchaseOrder.purchaseOrderNumber;
order.PoDate = purchaseOrder.orderDate;
order.PaymentTerm__c = purchaseOrder.paymentTerm;//???
order.PaymentMethod__c = purchaseOrder.paymentMethod; //???

PurchaseOrderInboundRequest.Address billingAddress
                                    = purchaseOrder.billingAddress;

order.BillingStreet = billingAddress.street;
order.BillingCity = billingAddress.city;
order.BillingState = billingAddress.state;
order.BillingPostalCode = billingAddress.postalCode;

PurchaseOrderInboundRequest.Address shippingAddress
                                    = purchaseOrder.shippingAddress;

order.ShippingStreet = shippingAddress.street;
order.ShippingCity = shippingAddress.city;
order.ShippingState = shippingAddress.state;
order.ShippingPostalCode = shippingAddress.postalCode;
```

Simples, não é mesmo? Porém, temos um ponto importante quando tratamos de integração entre dois sistemas. Geralmente os sistemas externos não conhecem os `Ids` da Salesforce, e devemos trabalhar com o código interno desses sistemas. Para tratar desses mapeamentos de códigos, utilizamos o conceito de chave externa nos objetos na plataforma.

Os campos definidos como chave externa possuem o código do **Sistema do Cliente**, que representa aquele registro no seu sistema. A imagem a seguir demonstra essa relação:

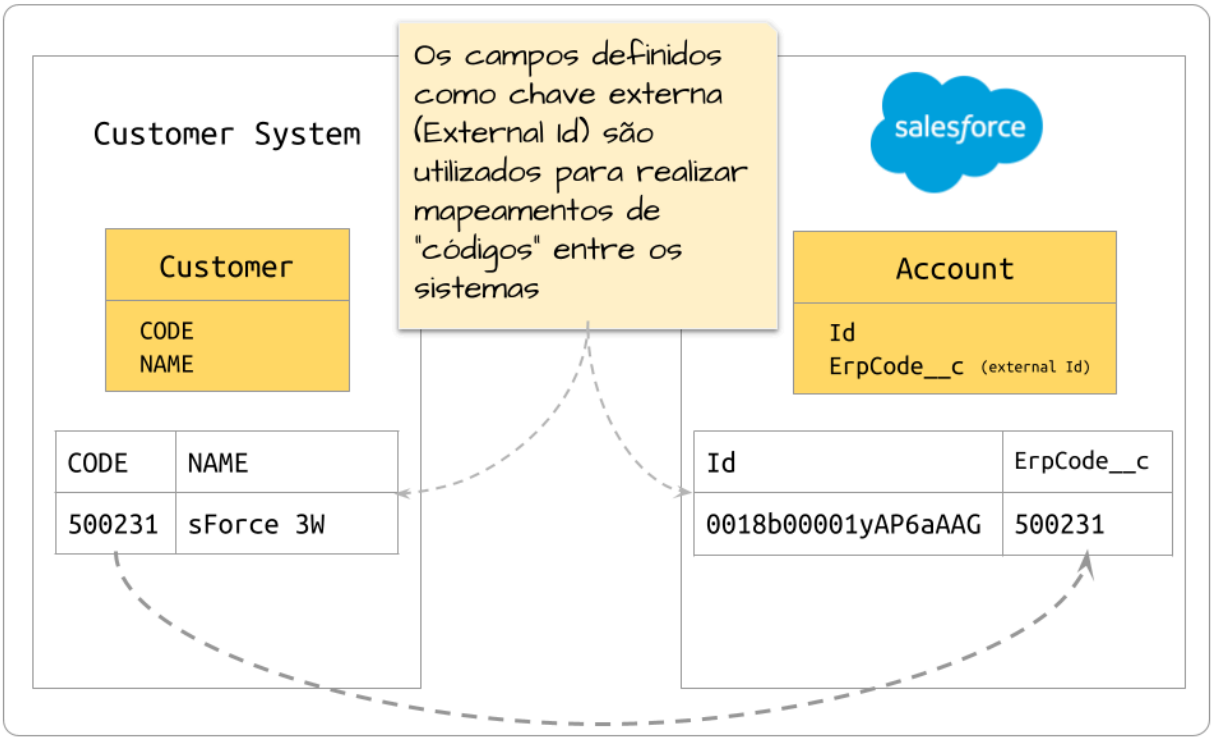


Figura 10.12: Mapeamento de chaves externas

Como podemos utilizar a chave externa em nosso mapeamento? Precisamos criar a instância do objeto em questão, informando o campo *chave*. Veja o exemplo a seguir, onde criamos uma instância de `Account` mapeando o campo `ErpCode__c`, que é uma chave externa:

```
Account account = new Account ( ErpCode__c
                                = purchaseOrder.accountCode);
```

#se-liga-na-info 😎

Vale lembrar que, para Objetos *Standard*, basta remover o `Id` do nome do campo; para Objetos *Customizados*, nós substituímos o `__c` por `__r`, para indicar que o campo é um relacionamento. Esse mecanismo de mapeamento deve ser utilizado em composição com o comando de DML de `upsert <SObject(s)> <External Id Field Name>`, caso contrário ocorrerá um erro ao salvar o registro.

Vamos então ajustar o nosso código para utilizar essa nova forma de atribuir os dados de `Account` em `Order`:

```
Order order = new Order();

order.Account = new Account ( ErpCode__c =
                                    purchaseOrder.accountCode);

order.PaymentTerm__r = new PaymentTerm__c (
                      ExternalId__c = purchaseOrder.paymentTerm);

order.PaymentMethod__r = new PaymentMethod__c (
                    ExternalId__c = purchaseOrder.paymentMethod);

// demais partes estão omitidas.
```

Agora que implementamos o mapeamento dos campos, vamos terminar a implementação do nosso método de criação de *Pedido*, `create (PurchaseOrderInboundRequest purchaseOrder)`:

```
PurchaseOrderInboundTransformer transformer =
                           new PurchaseOrderInboundTransformer();

Order order = transformer.toOrder(purchaseOrder);

List<OrderLineItem> orderItems = transformer.toOrderItems(
                                                  purchaseOrder );
```

```
OrderRepository orderRepository = new OrderRepository();

OrderItemRepository orderItemRepository
                                = new OrderItemRepository();

orderRepository.save ( order );
orderItemRepository.save ( orderItems );
```

Como fica a implementação da nossa classe `OrderRepository`? Vamos sobrescrever o método `save` da nossa classe `AbstractRepository`, redeterminando o comportamento da *classe-mãe*, para que a gente possa salvar os dados de `Order` utilizando o campo `ErpCode__c` como chave externa:

```
public class OrderRepository extends AbstractRepository {

    override
    public SObject save ( SObject order ) {

        upsert (Order) order ErpCode__c;

        return order;

    }

}
```

Essa implementação resolve o problema do fluxo do serviço que estamos implementando, mas deixa outras brechas. Pergunte-se: **sempre** que um `Order` for salvo, será utilizado o `upsert` com o campo `ErpCode__c` como chave externa? Provavelmente sua resposta é "não", pois a `OrderRepository` é uma classe que tem o propósito de ser reutilizada, e haverá outros cenários onde você utilizará o `Id` ou outra chave.

Esse é um cenário onde podemos deixar esse código mais genérico e elegante utilizando o método de `upsert` da classe `Database`, pois

ele recebe uma `List<SObject>` e um `Schema.SObjectField` que representa o campo que será utilizado como chave externa.

> **#fica-a-dica** 🤔
>
> **Conheça a sua API**
>
> Gosto de ressaltar que você deve sempre utilizar os recursos providos pela linguagem. A cada nova versão da API, visite o guia de referência para se atualizar das novas implementações e funcionalidades. Para isso, acesse: https://developer.salesforce.com/docs/atlas.en-us.apexref.meta/apexref/apex_ref_guide.htm.

"Bora" ajustar o nosso *code*! Como desejamos ser mais genéricos, vamos alterar a `AbstractRepository`, adicionando o método `save`, que recebe dois argumentos, `List<SObject>` e um `Schema.SObjectField`:

```
virtual
public  List<SObject> save ( List<SObject> records
                           , Schema.SObjectField field ) {

    if (records == null || records.isEmpty()) {
        return new List<SObject>();
    }

    Database.upsert(records, field);

    return records;
}
```

Agora vamos consumir esse novo método. Para isso, precisamos obter o `Schema.SObjectField` do campo `ErpCode__c` através da classe `Schema`, ficando assim:

```
Schema.SObjectField ORDER_EXTERNAL_ID = Schema.Order
                                            .ErpCode__c
                                            .getDescribe()
                                            .getSObjectField();

// outras partes estão omitidas.

orderRepository.save ( order , ORDER_EXTERNAL_ID );
```

Excelente! Resolvemos o `save` de `Order`, porém ainda precisamos ajustar o `save` de `OrderLineItem`, pois ele necessita não somente da chave externa para realizar a operação, mas também mapear o `Id` de `Order` para o campo `OrderId` da classe `OrderLineItem`.

#se-liga-na-info 😎

Os objetos `Order` e `OrderLineItem` possuem uma relação chamada de **Mestre-Detalhe**, que determina que só pode haver um `OrderLineItem` se houver um `Order`. Essa relação **Mestre-Detalhe** também determina a visibilidade de registros, onde o **Detalhe** herda o compartilhamento do **Mestre**, que, no nosso cenário, é determinado pelo objeto `Order`.

Voltando para a nossa implementação, vamos refatorar a nossa classe `OrderLineItemRepository`, que deverá salvar os itens do *Pedido*. Esse novo método vai encapsular a determinação do `OrderId` em `OrderItem` e salvar os itens logo em seguida:

```
public class OrderItemRepository extends AbstractRepository {

    Schema.SObjectField ORDER_ITEM_EXTERNAL_ID =
                                    Schema.OrderItem.ErpCode__c
                                        .getDescribe()
                                        .getSObjectField();

    virtual
```

```
    public List<OrderItem> save ( Order order
                                , List<OrderItem> orderItems ) {

        for (OrderItem orderItem : orderItems) {

            // em caso de update não podemos alterar o valor do
            // campo MasterDetail
            if (orderItem.OrderId != null) continue;

            orderItem.OrderId = order.Id;

        }

        save (orderItems, ORDER_ITEM_EXTERNAL_ID);

        return orderItems;
    }
}
```

Essa implementação é interessante e pode ser utilizada em diversos contextos, mas podemos utilizar o poder dos *ids* externos a nosso favor. Assim como mapeamos os campos de `PaymentTerm__c` e `AccountId` em `Order`, podemos aplicar a mesma estratégia ao campo `OrderId` de `OrderItem`. Veja parte da implementação do método `toOrderItems` a seguir:

```
public List<OrderItem> toOrderItems (
                    PurchaseOrderInboundRequest purchaseOrder) {

    List<OrderItem> items = new List<OrderItem>();

    for (PurchaseOrderInboundRequest.PurchaseItem item:
                                        purchaseOrder.items ) {

        OrderItem orderItem = new OrderItem ();

        orderItem.Order = new Order (
            ErpCode__c = purchaseOrder.purchaseOrderNumber
        );
```

```
        // parte do código foi omitida
        items.add (orderItem);

    }

    return items;
}
```

Com isso, deixamos a nossa etapa de persistência mais simples, utilizando somente o método `save` padrão da `AbstractRepository`. Essas variações de implementação são importantes para que você tenha mais repertório e, com isso, maior capacidade de entregar códigos elegantes.

Com tudo isso em mãos, nosso código está ficando cada vez mais profissional, mas vamos fazer mais algumas refatorações para deixá-lo ainda mais elegante. Começando pela `OrderController`, vamos mover sua implementação para uma classe chamada `OrderInboundService`. Veja a seguir a implementação da `OrderInboundService`:

```
public class OrderInboundService {

    OrderRepository orderRepository;
    OrderItemRepository orderItemRepository;
    OrderSubItemRepository orderSubItemRepository;

    PurchaseOrderInboundTransformer transformer;

    public OrderInboundService () {

        this.transformer = new PurchaseOrderInboundTransformer();

        this.orderRepository = new OrderRepository();

        this.orderItemRepository = new OrderItemRepository();

        this.orderSubItemRepository
```

```
                                = new OrderSubItemRepository();
    }


    public PurchaseOrderInboundResponse save (
                    PurchaseOrderInboundRequest purchaseOrder ) {


        Order order = transformer.toOrder(purchaseOrder);

        List<OrderItem> orderItems =
                     transformer.toOrderItems( purchaseOrder );

        List<OrderSubItem__c> orderSubItems =
                    transformer.toOrderSubItems( purchaseOrder );

        this.orderRepository.save(order);
        this.orderItemRepository.save (orderItems);
        this.orderSubItemRepository.save (orderSubItems);

        return new PurchaseOrderInboundResponse( order.Id );

    }

}
```

Agora, ajustando a `OrderController` :

```
@RestResource (urlMapping='/api/orders/*')
global class OrderController {

    public static OrderInboundService orderInboundService;

    static {
        orderInboundService = new OrderInboundService();
    }

    @HttpPost
    global static PurchaseOrderInboundResponse create (
                    PurchaseOrderInboundRequest purchaseOrder ) {
```

```
        return orderInboundService.save(purchaseOrder);

    }

}
```

Com todos esse ajustes, chegamos a uma implementação mais robusta. Perceba que, com cada etapa da implementação, todos os níveis da aplicação foram ajustados para que o fluxo ficasse com um padrão de código mais limpo e organizado.

Você pode ver o código completo no repositório do projeto: https://github.com/ercarval/salesforce-book-snippets.

Vamos colocar mais tempero em nosso cenário

O cenário de integração de *Pedido de Compra* é considerado um dos mais críticos para operação de diversos clientes. Isso porque ele pode impactar diretamente na receita da empresa. De forma geral, integrações sistêmicas necessitam de alta capacidade de rastreabilidade, seja do lado que enviou a requisição ou do lado que a recebeu.

Esse é um cenário em que não podemos contar somente com o log gerado pela plataforma. Necessitamos manter esses dados por mais tempo, gerenciando o seu tempo de retenção. Para atender a esse requisito não funcional, vamos realizar um *Log* da integração realizada, indicando se teve sucesso ou falha, e gravar o *payload* recebido.

Para isso, vamos criar um novo objeto chamado `IntegrationLog__c`, que deverá ter alguns dados básicos da integração, como nome da integração, status, *payload*.

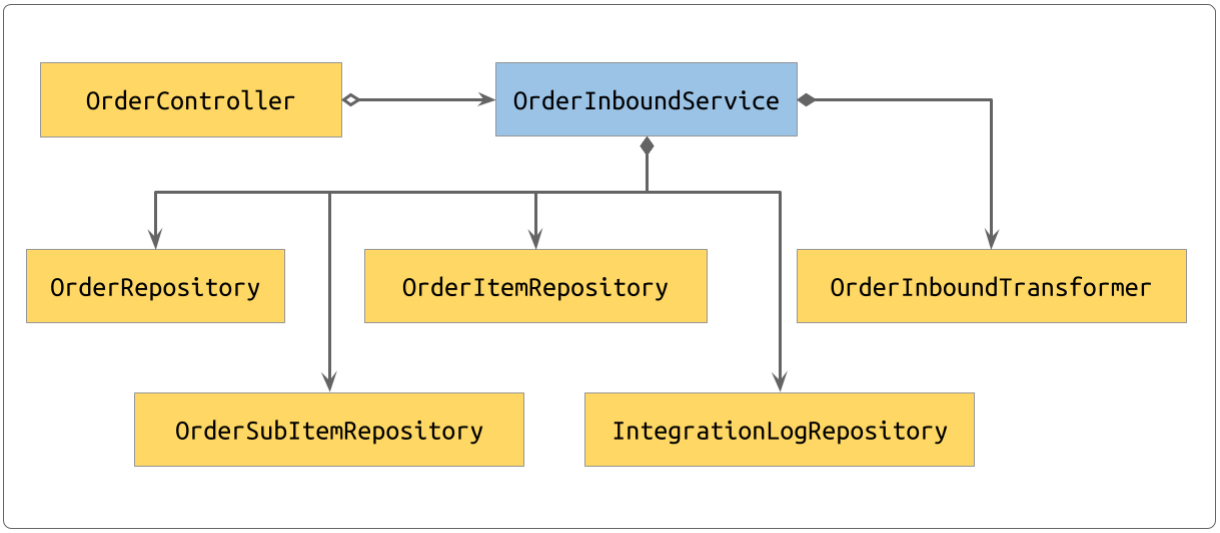


Figura 10.16: Order Inbound Service

Vamos ajustar a classe `OrderInboundService` para atender a essa necessidade, adicionado os métodos de `log ( PurchaseOrderInboundRequest )` e `log ( PurchaseOrderInboundRequest, Exception )`:

```
public PurchaseOrderInboundResponse save (
                  PurchaseOrderInboundRequest purchaseOrder ) {
    Order order;
    try {

        order = transformer.toOrder(purchaseOrder);

        List<OrderItem> orderItems =
                    transformer.toOrderItems( purchaseOrder );

        List<OrderSubItem__c> orderSubItems =
                transformer.toOrderSubItems( purchaseOrder );

        this.orderRepository.save(order);
        this.orderItemRepository.save (orderItems);
        this.orderSubItemRepository.save (orderSubItems);

        log (purchaseOrder);
```

```
    } catch (DmlException e) {
        log (purchaseOrder, e);
        throw e;
    } catch (Exception e) {
        log (purchaseOrder, e);
        throw e;
    }
    return new PurchaseOrderInboundResponse( order.Id );

}
void log ( PurchaseOrderInboundRequest purchaseOrder ) {
    IntegrationLog__c log = transformer.toLog (purchaseOrder);
    this.integrationLogRepository.save (log);
}
void log ( PurchaseOrderInboundRequest purchaseOrder,
                   DmlException dmlException ) {
    IntegrationLog__c log = transformer.toLog ( purchaseOrder ,
                                                dmlException );
    this.integrationLogRepository.save (log);

}
void log ( PurchaseOrderInboundRequest purchaseOrder,
```

```
            Exception ex ) {

    IntegrationLog__c log =
                    transformer.toLog ( purchaseOrder ,
                                                ex );

    this.integrationLogRepository.save (log);

}
```

No código anterior, estamos logando tanto no sucesso quanto na falha, por isso os métodos de `log` são executados no `catch`. Mas vamos focar mais no fluxo de erro.

As falhas podem ocorrer de diversas formas, seja por um erro de implementação ou por um conteúdo inválido enviado. Para o primeiro caso, devemos mitigar com testes de unidade; já para o segundo, devemos somente reportar o problema e, agora, ter capacidade de rastrear a informação.

Um exemplo de problema que pode ocorrer é a dessincronização de dados mestres. Imagine que o sistema de origem criou uma nova forma de pagamento (`paymentMethod`) e esse dado não foi mapeado no **Salesforce**, no objeto `PaymentMethod__c`. Esse tipo de problema geraria uma `DmlException`, indicando que não foi possível encontrar a informação para o objeto `PaymentMethod__c` com a chave externa informada. Mas, como implementamos o nosso `log` de integração, não teremos problemas em identificar o erro e corrigi-lo.

Então, após a execução de um cenário de falha, devemos possuir registros no `IntegrationLog__c`, correto? Vamos realizar a consulta a seguir para observar o log:

```
SELECT Id, Name, ReceivePayload__c, CreatedDate FROM IntegrationLog__c
LIMIT 1
```

A consulta retornou zero resultado, ou seja, não foi possível encontrar o log! Mas como isso é possível? Nós colocamos os métodos de `save` no `catch` e mesmo assim ele não salvou o

registro! Esse comportamento ocorre devido à forma como o **Salesforce** realiza a gestão do fluxo transacional. Vamos entender primeiro o que é esse nome bonito, "fluxo transacional", para depois resolver o nosso problema de logs não gerados.

10.5 Entendendo o ciclo transacional

Fluxo Transacional é o processo pelo qual a **Salesforce** garante que todas as operações (DMLs) realizadas no seu banco de dados possam ser confirmadas ou desfeitas, caso tenha algum problema durante a sua execução. Em outras palavras, é quando é realizado o *commit* ou *rollback* de uma transação, garantindo a integridade dos dados.

Em nosso primeiro cenário de negócio, apresentado anteriormente, tínhamos a necessidade de armazenar os dados de três objetos (`Order` , `OrderLineItem` , `OrderSubItem__c`); então, para que o *Pedido* possa ser considerado "criado", o fluxo transacional deve garantir que todos os dados sejam armazenados com sucesso; caso contrário, o pedido em questão **não** deverá ser criado.

Vamos ver graficamente quando ocorre um erro durante o fluxo de execução:

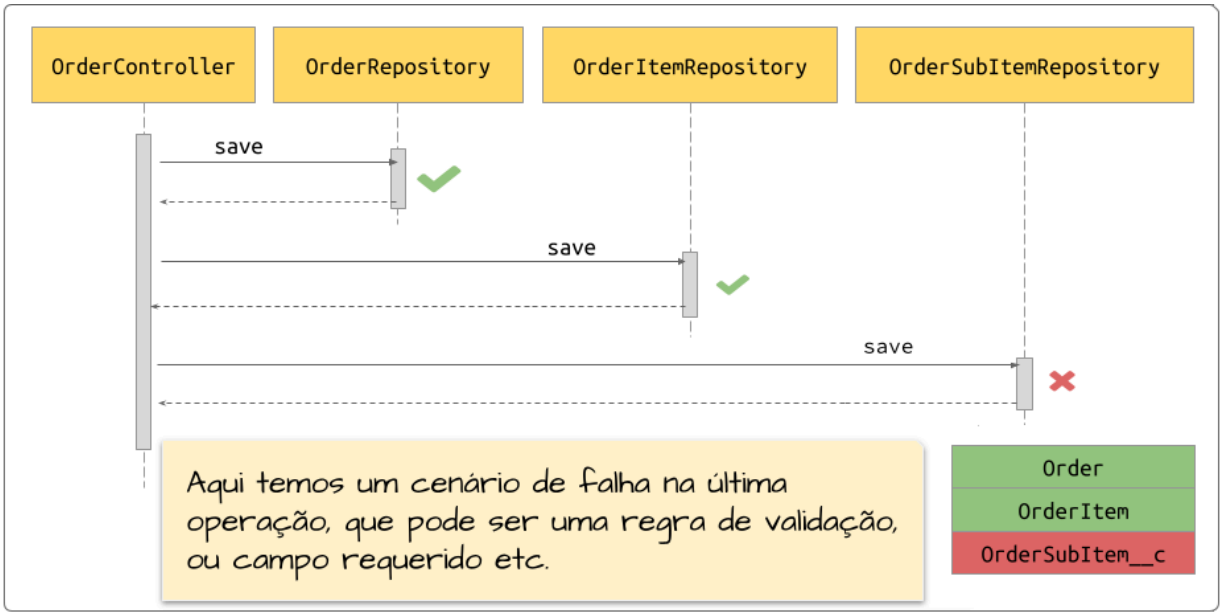


Figura 10.17: Fluxo de execução com erro

O que acontece caso ocorra um erro durante a execução? Nesse caso de erro, todas as operações realizadas no banco serão desfeitas, todos os dados armazenados anteriormente serão perdidos. Isso ocorre devido ao mecanismo de garantia transacional da plataforma, que aplica as propriedades **ACID**:

- **Atomicidade:** cada instrução em uma transação (ler, escrever, atualizar ou excluir dados) é tratada como uma única unidade. Cada uma deve ser executada de forma total, ou nenhuma é executada. Essa propriedade impede a ocorrência de perda e corrupção de dados.
- **Consistência:** garante que as transações apenas façam alterações nos objetos de maneiras predefinidas e previsíveis. A consistência transacional garante que corrupção ou erros em seus dados não criem consequências não intencionais para a integridade de seu objeto;
- **Isolamento:** Quando vários usuários estão lendo e gravando no mesmo objeto ao mesmo tempo, o isolamento de suas transações garante que as transações simultâneas não interfiram ou afetem umas às outras. Cada solicitação pode

ocorrer como se estivesse ocorrendo uma a uma, mesmo que, na verdade, estejam ocorrendo simultaneamente;
- **Durabilidade:** Garante que as alterações em seus dados feitas por transações executadas com sucesso serão salvas, mesmo em caso de falha do sistema.

A Salesforce inicializa o fluxo transacional a partir da requisição recebida e o finaliza quando retorna a resposta para *Cliente*, seja um "Browser", ou uma chamada de um serviço como o apresentado na imagem a seguir:

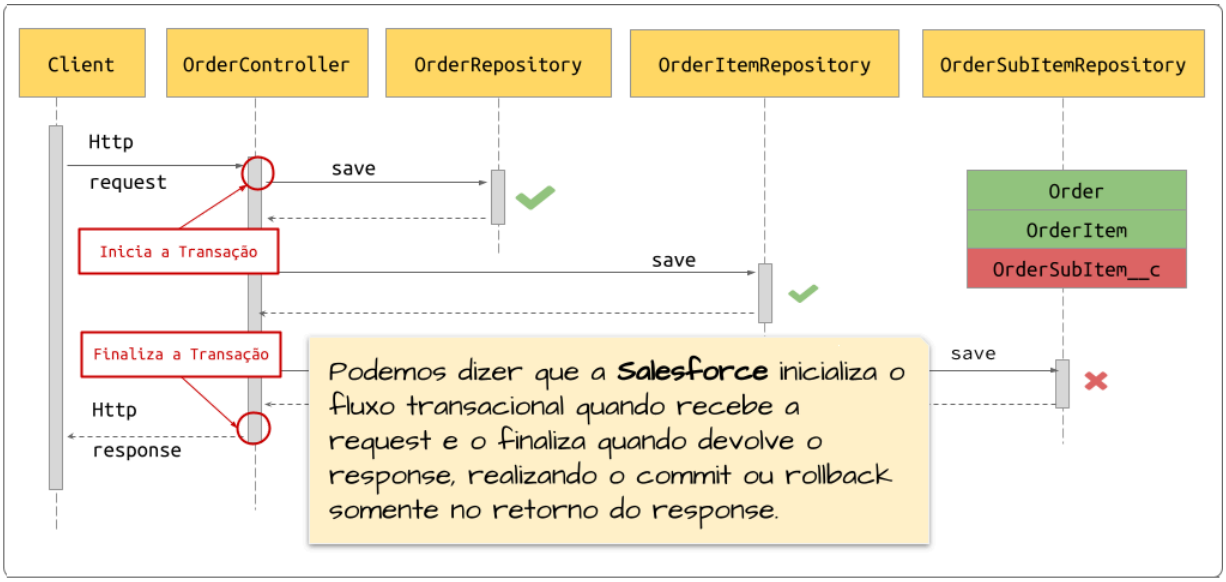


Figura 10.18: Fluxo de execução completo

Mas como corrigimos a nossa gestão de log?

Voltando para o nosso problema de salvar os dados de log, podemos ver que ele ocorre porque estamos propagando a *exception* para *Cliente*, como o apresentado na imagem anterior; dessa forma, todos os dados são descartados.

Para contornar esse problema, devemos refatorar a classe `OrderController` , para que, em vez de retornar uma falha, retorne um código HTTP 400. Dessa forma, o fluxo transacional não recebe uma `Exception` e não realiza o *rollback* da transação.

Nessa refatoração, não podemos mais definir o tipo de retorno como `PurchaseOrderInboundResponse`, pois o APEX não conseguiria retornar um tipo diferente do determinado pelo método `create` da `OrderController`. Então vamos fazer o controle do *response* na "mão". Vamos definir o método como `void` e gerir o conteúdo do `RestResponse`.

Veja os passos da nossa refatoração:

1. Ajustar o método para `void`;
2. Configurar o `Content-Type` para `application/json`;
3. Definir o código padrão de retorno para `200`;
4. Em caso de falha, retornar erro `400`.

```
@HttpPost
global static void create (
                PurchaseOrderInboundRequest purchaseOrder ) {

    RestResponse response = RestContext.response;

    response.statusCode = 200;
    response.headers.put('Content-Type'
                        , 'application/json');

    String responsePayload;

    try {

        PurchaseOrderInboundResponse purchaseOrderResponse;

        purchaseOrderResponse = orderInboundService.save(
                                                purchaseOrder);

        responsePayload = purchaseOrderResponse.toJson();

    } catch (DmlException e) {

        response.statusCode = 400;
```

```
            responsePayload = '{ "error" : "'
                                    + e.getDmlMessage(0)
                                    + '" }';

        } catch (Exception e) {

            response.statusCode = 400;

            responsePayload = '{ "error" : "'
                                    + e.getMessage()
                                    + '" }';

        } finally {

            response.responseBody = Blob.valueOf(responsePayload);

        }

}
```

Nossa! Como ficou grande a nossa implementação, e que solução deselegante, retornar uma mensagem de erro como uma `String` para o serviço de integração. Vamos então refatorar a nossa refatoração :).

1. Criar uma classe de erro padrão `RestResponseError` ;
2. Criar uma classe `RestResponseBuilder` para facilitar a manipulação do `RestResponse` .

```
public class RestResponseError {

    public String error;

    public RestResponseError ( DmlException dmlException ) {
        this ( dmlException.getDmlMessage(0) );
    }

    public RestResponseError ( Exception ex ) {
        this ( ex.getMessage() );
```

```
    }

    public RestResponseError ( String error ) {
        this.error = error;
    }

}

public class RestResponseBuilder {

    RestResponse response;

    public RestResponseBuilder (RestResponse response) {
        this.response = response;
        this.response.statusCode = 200;
        this.response.headers.put('Content-Type'
                                  , 'application/json');
    }

    public void withBody ( Object body ) {
        this.response.responseBody = Blob.valueOf(
                                       JSON.serialize(body) );
    }

    public void withError ( DmlException dmlException ) {
        this.response.statusCode = 400;
        withBody ( new RestResponseError ( dmlException ) );
    }

    public void withError ( Exception ex ) {
        this.response.statusCode = 400;
        withBody ( new RestResponseError ( ex ) );
    }

}

@HttpPost
global static void create (
                PurchaseOrderInboundRequest purchaseOrder ) {
```

```
    RestResponseBuilder responseBuilder =
                    new RestResponseBuilder(RestContext.response);

    try {

        PurchaseOrderInboundResponse purchaseOrderResponse;

        purchaseOrderResponse = orderInboundService.save(
                                                purchaseOrder);

        responseBuilder.withBody(purchaseOrderResponse);

    } catch (DmlException e) {

        responseBuilder.withError(e);

    } catch (Exception e) {

        responseBuilder.withError(e);

    }

}
```

Agora melhorou bastante a nossa implementação! Podemos retornar diferentes tipos e podemos controlar qual o tipo de HTTP Code queremos retornar.

Vamos testar novamente com o mesmo método de pagamento (`paymentMethod`) inválido. Devemos receber um Http Code `400` com a nova mensagem de erro. Agora é a hora da verdade! Vamos fazer uma nova *query* em `IntegrationLog__c` e ver se temos um registro salvo:

```
SELECT Id, Name, ReceivePayload__c, CreatedDate FROM IntegrationLog__c
LIMIT 1
```

Sucesso! Nosso log foi salvo, mas essa nossa implementação deixou um buraco... Gostaria que você refletisse por alguns segundos, antes de continuar a leitura.

Pensou? Como nós driblamos o mecanismo de *rollback* padrão, ele não desfaz as operações bem-sucedidas. Então se ocorrer um erro ao salvar `OrderSubItem__c` , os outros objetos permanecerão salvos, quebrando o princípio da atomicidade da transação.

Com esse tipo de ajuste, precisamos realizar o *rollback* da transação manualmente. Mais um ajuste! Vamos utilizar a `Database.setSavepoint()` para criar um segmento de *rollback*, e todas as operações que forem realizadas logo após a definição do `savePoint` poderão ser desfeitas caso ocorra algum erro. Vamos então refatorar o `OrderInboundService` para utilizar o `Database.setSavepoint()` para criar um segmento de *rollback*:

```
SavePoint savePoint = Database.setSavePoint();

try {

    Order order = transformer.toOrder(purchaseOrder);

    List<OrderItem> orderItems =
                transformer.toOrderItems( purchaseOrder );

    List<OrderSubItem__c> orderSubItems =
            transformer.toOrderSubItems( purchaseOrder );

    this.orderRepository.save(order);
    this.orderItemRepository.save (orderItems);
    this.orderSubItemRepository.save (orderSubItems);

    log (purchaseOrder);

} catch (DmlException e) {

    Database.rollback(savePoint);
```

```
    log (purchaseOrder, e);

    throw e;

} catch (Exception e) {

    Database.rollback(savePoint);

    log (purchaseOrder, e);

    throw e;

}
```

Basicamente adicionamos mais três linhas em nossa implementação, e os blocos `catch` receberam o `Database.rollback(savePoint)` para desfazer as operações caso ocorra algum erro. Agora podemos rodar tudo novamente! **Sucesso!**

> Esse é um exemplo muito próximo do real. Esse tipo de implementação poderia ser utilizado em qualquer aplicação produtiva.

Tem como deixar ainda mais elegante? A resposta é **sim**: poderíamos ter criado uma classe que encapsulasse `Order`, `OrderLineItem` e `OrderSubItem__c` e, com isso, nosso `Transformer` passaria a retornar o objeto encapsulado, não mais cada objeto isolado. E para fechar, teríamos um método `save` que recebe a nova classe como parâmetro. Mas isso eu deixo para você exercitar :).

Você tem agora ampla capacidade de realizar integrações sistêmicas utilizando o protocolo HTTP de forma elegante e escalável. Este capítulo trouxe bastante código, com uma sequência evolutiva, fechando com um código produtivo de fato. Aplicamos várias técnicas abordadas anteriormente para realizar as refatorações, sempre observando o Princípio da Responsabilidade Única, evitando ao máximo repetição de código, sem utilizar `ifs`.

O próximo capítulo trará mais um conjunto de elementos para sua caixa de ferramentas. Ele completa o cenário de integração sistêmica com a capacidade de realizar integrações assíncronas através de *Triggers*. Tudo conectado!

Parte 4: Trabalhando de forma assíncrona

> "*A lagarta não precisa de um milagre para virar uma borboleta, ela precisa de um processo. Os processos, da vida, são muito parecidos com um casulo. Ela entra no casulo em um processo doloroso, escuro e solitário, mas somente através deste ela consegue se transformar em uma borboleta e dar voos altos na vida. Nós ficamos o tempo todo fugindo dos processos que nos são apresentados. Dias de escuridão, solidão e aperto são a única forma de você dar voos mais altos.*"
>
> — Tiago Brunet (@tiagobrunet)

Muitas vezes nos deparamos com a necessidade de realizar tarefas em paralelo, ou seja, mais de uma atividade ao mesmo tempo. Em software não é diferente: muitas vezes precisamos realizar operações em paralelo, as quais chamaremos de *assíncronas*. Abordaremos nesta etapa as várias alternativas da linguagem APEX para tratarmos os processos assíncronos.

Capítulo 11
Alternativas de processamento assíncrono

O processamento assíncrono e síncrono são dois modelos fundamentais de programação que têm impacto direto na eficiência e no desempenho de um programa. Compreender a diferença entre esses modelos é crucial para o desenvolvimento de sistemas eficientes e escaláveis.

Processamento síncronos são aqueles que são executados de forma linear. Cada operação precisa ser finalizada para que a próxima seja executada. Isso significa que a execução do programa

é bloqueada quando uma tarefa está sendo executada e só continua após sua conclusão. Observe a imagem a seguir:

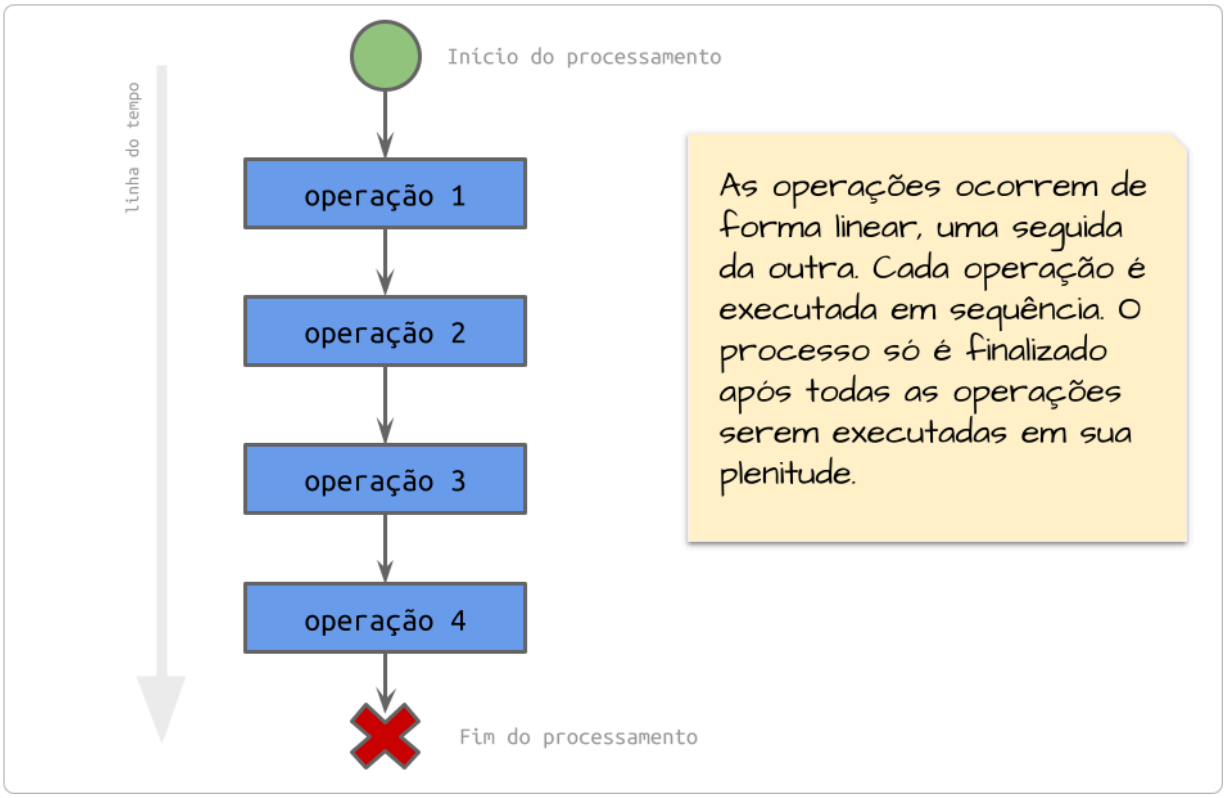


Figura 11.1: Processamento síncrono

Por outro lado, no **processamento assíncrono**, as operações são executadas independentemente umas das outras e, portanto, não precisam esperar que a operação anterior seja concluída antes de iniciar sua própria execução. Isso permite que a plataforma realize outras operações enquanto espera que o processamento assíncrono seja concluído, o que aumenta a eficiência e o desempenho.

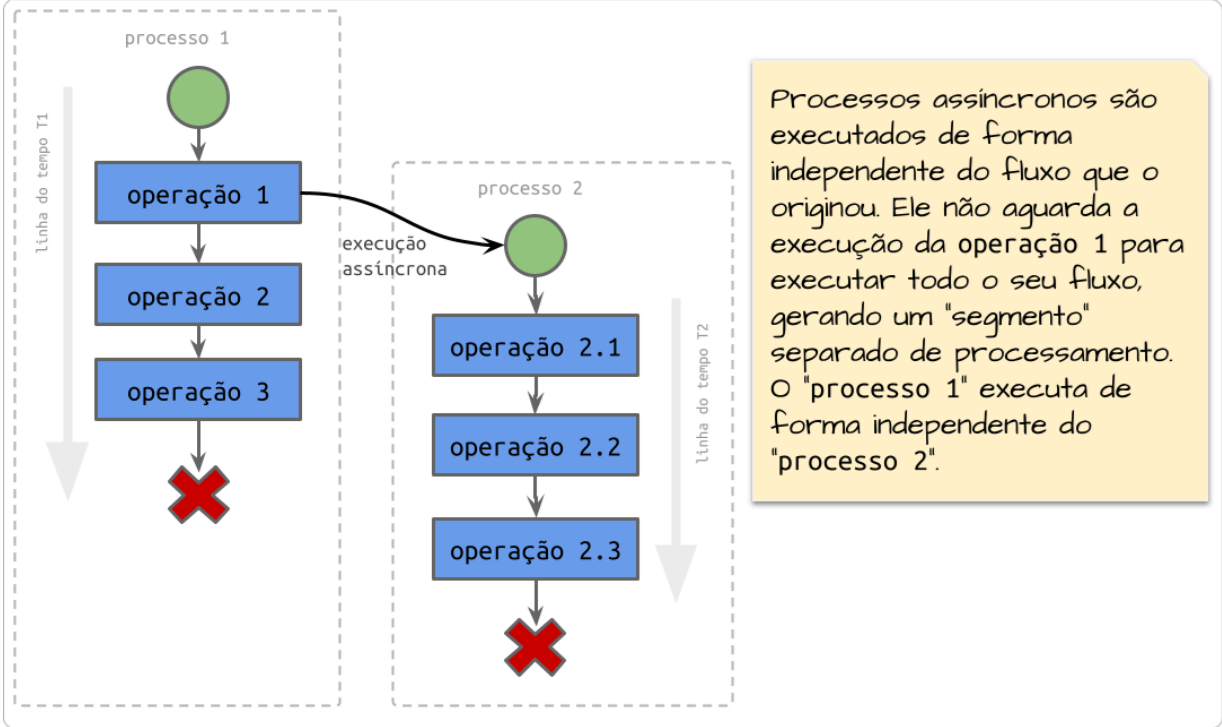


Figura 11.2: Processamento assíncrono

Com isso, o processamento assíncrono é um recurso poderoso que deve ser utilizado pelos desenvolvedores para otimizar seus sistemas e obter melhores resultados. Esses benefícios do processamento assíncrono também são refletidos no aumento dos limites permitidos pela plataforma. Veja a tabela a seguir:

Limite	Síncrono	Assíncrono
Total de consultas SOQL	100	200
Tamanho total da memória *heap*	6 MB	12 MB
Tempo máximo gasto de CPU nos servidores da Salesforce	10 segundos	60 segundos

Para saber mais sobre os limites da plataforma, acesse: https://developer.salesforce.com/docs/atlas.en-us.salesforce_app_limits_cheatsheet.meta/salesforce_app_limits_cheatsheet/salesforce_app_limits_platform_apexgov.htm.

Vamos então ver as alternativas para trabalhar com processos assíncronos.

11.1 Trabalhe com chamadas futuras

Chamadas futuras são uma das alternativas para trabalhar com processamento assíncrono. A seguir, temos alguns exemplos de negócio em que podemos trabalhar com processos assíncronos e, consequentemente, com chamadas futuras:

- Enviar e-mails no final de um processamento;
- Enviar uma notificação (*Custom Notification*);
- Realizar integração com outro sistema após uma atualização de dados;
- Executar processos com alto custo de processamento.

Vamos começar com uma operação muito simples: gravar em um objeto de log alguns dados de forma assíncrona. Ele armazenará uma mensagem e os dados que estavam sendo processados naquele momento. Esse exemplo é apenas para ilustrar o uso de chamadas futuras; nele, vamos criar a classe `AsyncLogger` e utilizá-la na classe de `AccountTriggerHandler`.

Para utilizar chamadas futuras, precisamos implementar um método que obrigatoriamente precisa ser `public static void` e utilizar a anotação `@future`:

```
public class AsyncLogger {
```

```
    @future
    public static void log ( String message
                           , String payload ) {

        AsyncLoggerRepository repository
                                = new AsyncLoggerRepository();

        Log__c log = new Log__c ( Message__c = message
                                , Payload__c = payload );

        repository.save ( log );

    }

}

public with sharing class AccountTriggerHandler
                                        extends TriggerHandler {

    // parte da implementação será omitida

    override
    public void afterInsert() {

        log ('AccountTrigger afterInsert assíncrono');

    }

    override
    public void afterUpdate() {

        log ('AccountTrigger afterUpdate assíncrono');

    }

    void log (String message) {

        AsyncLogger.log ( message
```

```
                    , JSON.serialize ( this.newAccounts ) );

    }

}
```

#se-liga-na-info 😎

Métodos anotados com `@future` só podem receber parâmetros "primitivos" como `String`, `Integer`, `Id`. Eles também podem receber uma coleção, como `List`, `Set` e `Map`, de um tipo primitivo.

Simplão! Para criar uma *Thread*, uma nova chamada futura, basta chamar o método anotado com `@future` e a plataforma iniciará um novo processo assíncrono. A imagem a seguir ilustra o ciclo de vida de uma chamada futura:

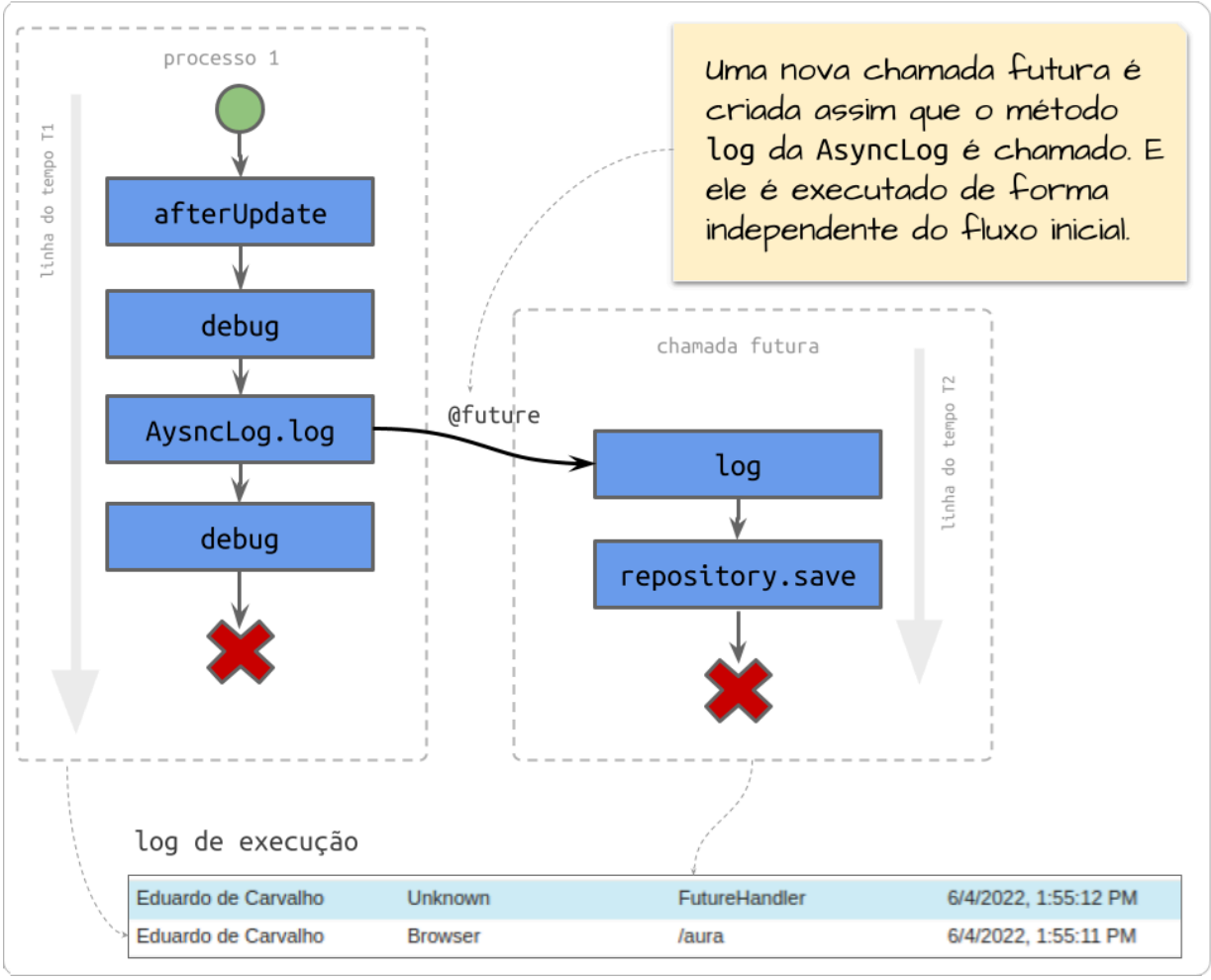


Figura 11.4: Ciclo de execução de uma chamada futura

#se-liga-na-info 😎

Apesar de a plataforma aumentar os limites de processamento, ela também adicionou outros, como não ser possível realizar uma nova chamada futura dentro de uma chamada futura. Imagine que nosso método `AsyncLog.log` chame outro método também anotado com `@future`; ele estará tentando criar uma nova chamada futura dentro de uma já existente.

11.2 Testando chamadas futuras

Vamos criar a classe `AsyncLoggerTest`, que vai executar uma chamada ao método `AsyncLogger.log` e verificar se foi registrada uma nova entrada no objeto `Log__c`:

```
@isTest
public class AsyncLoggerTest {

    @isTest
    public static void shouldLogMessageWithPayload() {

        String expectedMessage =
                            'Test async call in test execution';

        AsyncLogger.log ( expectedMessage
                        , '{"Name": "testPayload"}' );

        Log__c log = [ SELECT Id, Message__c, Payload__c
                        FROM Log__c LIMIT 1];

        System.assert( log != null );

        System.assertEquals( expectedMessage
                                ,  log.Message__c );

    }

}
```

Ao executar esse teste, recebemos um erro de `System.QueryException: List has no rows`, justamente na consulta do objeto `Log__c`. Isso ocorre porque o processo assíncrono não foi realizado.

```
=== Apex Code Coverage for Test Run 7078c00006s3ypF
TEST NAME                                CLASS BEING TESTED  OUTCOME  PERCENT  MESSAGE                                                           RUNTIME (MS)
───────────────────────────────────────  ──────────────────  ───────  ───────  ────────────────────────────────────────────────────────────────  ────────────
AsyncLogTest.shouldLogMessageWithPayload  AsyncLog           Fail     0%       System.QueryException: List has no rows for assignment to SObject
                                                                               Class.AsyncLogTest.shouldLogMessageWithPayload: line 13, column 1  31
```

Figura 11.6: Execução com erro de testes

Para executar operações assíncronas em classes de testes, precisamos utilizar o `Test.startTest()` juntamente com `Test.stopTest()` . Logo que o `Test.stopTest()` for executado, todos os processos assíncronos serão executados.

```
@isTest
public class AsyncLoggerTest {

    @isTest
    public static void shouldLogMessageWithPayload() {

        String expectedMessage =
                            'Test async call in test execution';

        Test.startTest();

        AsyncLogger.log ( expectedMessage
                        , '{"Name": "testPayload"}' );

        Test.stopTest();

        Log__c log = [ SELECT Id, Message__c, Payload__c
                        FROM Log__c LIMIT 1];

        System.assert( log != null );

        System.assertEquals( expectedMessage
                                , log.Message__c );

    }

}
```

Assim que envolvemos a execução do `AsyncLogger.log` com `Test.startTest()` e `Test.stopTest()`, o processo assíncrono será executado e, com isso, o dado é inserido e a *query* é realizada com sucesso!

```
=== Apex Code Coverage for Test Run 7078c00006s40z3
TEST NAME                                  CLASS BEING TESTED     OUTCOME  PERCENT  MESSAGE  RUNTIME (MS)
─────────────────────────────────────────  ─────────────────────  ───────  ───────  ───────  ────────────
AsyncLogTest.shouldLogMessageWithPayload   AsyncLog               Pass     100%              107
AsyncLogTest.shouldLogMessageWithPayload   AsyncLoggerRepository  Pass     100%              107
```

Figura 11.7: Teste realizado com sucesso

11.3 Cenário real com chamadas futuras

Agora que vimos um cenário simples de execução, vamos ver um outro mais próximo do mundo real. Ele envolverá a necessidade de enviar dados para um sistema externo a partir de uma alteração realizada em um objeto. Essa é a necessidade da Bisso Capital Inc., que deseja enviar todos os *Pedidos* aprovados na Salesforce para o seu sistema interno.

Esse é um cenário de negócio muito comum, que unifica diversas técnicas e implementações que abordamos em capítulos anteriores, como a utilização de `TriggerHandler`, `Filter`, `OutboundServices`, `Repository`. Mas ele tem uma pegadinha, que não foi abordada no capítulo de chamadas externas de propósito: não é possível realizar uma chamada externa com uma transação pendente, ou seja, com uma operação DML pendente.

Vou explicar melhor esse problema. Imagine quanto custaria de memória, espaço em disco, para um sistema com milhares de operações por segundo iniciar uma atualização no banco e realizar uma chamada externa, que pode levar até 120 segundos. Isso poderia aumentar em alguns petabytes de memória em disco só para suportar toda essa carga, pois envolve a criação de segmentos

de *rollback*, que ficam esperando por um sistema externo para serem confirmados ou desfeitos.

Com essa visão de custo operacional, até faz sentido essa limitação de não realizar *Callouts* com operações de DML pendente. Mas como "driblamos" esse limite? É aí que entram as chamadas assíncronas, que criam um novo fluxo independente da operação DML executada. Para implementar esse novo cenário de negócio, vamos quebrá-lo em partes.

Primeiramente devemos obter os pedidos aprovados. Eles serão obtidos através da *Trigger* em `Order`, que utilizará a *TriggerHandler* `OrderTriggerHandler`. Agora que temos todos as atualizações de pedidos passando pela nossa *TriggerHandler*, podemos filtrar somente os pedidos que foram modificados para *Aprovado*, através da classe *Filter* `OrderFilter`.

Nesse ponto temos todos os pedidos que foram aprovados, só precisamos cuidar do envio do pedido. Então vamos criar as classes de envio do *Pedido* propriamente, como `OrderOutboundService` e `OrderTransformer`, que farão a requisição HTTP para o sistema interno.

Por fim, podemos juntar todas as partes, e é aí que vem "o pulo do gato", onde aplicaremos o conhecimento adquirido de **utilização de chamadas futuras** para chamar a classe `OrderOutboundService` sem impactar o fluxo transacional. Nessa etapa implementaremos a classe `AsyncOrderOutboundService`, que basicamente terá um método `@future` que será o responsável por chamar a classe `OrderOutboundService`.

Tudo isso está representado no diagrama a seguir:

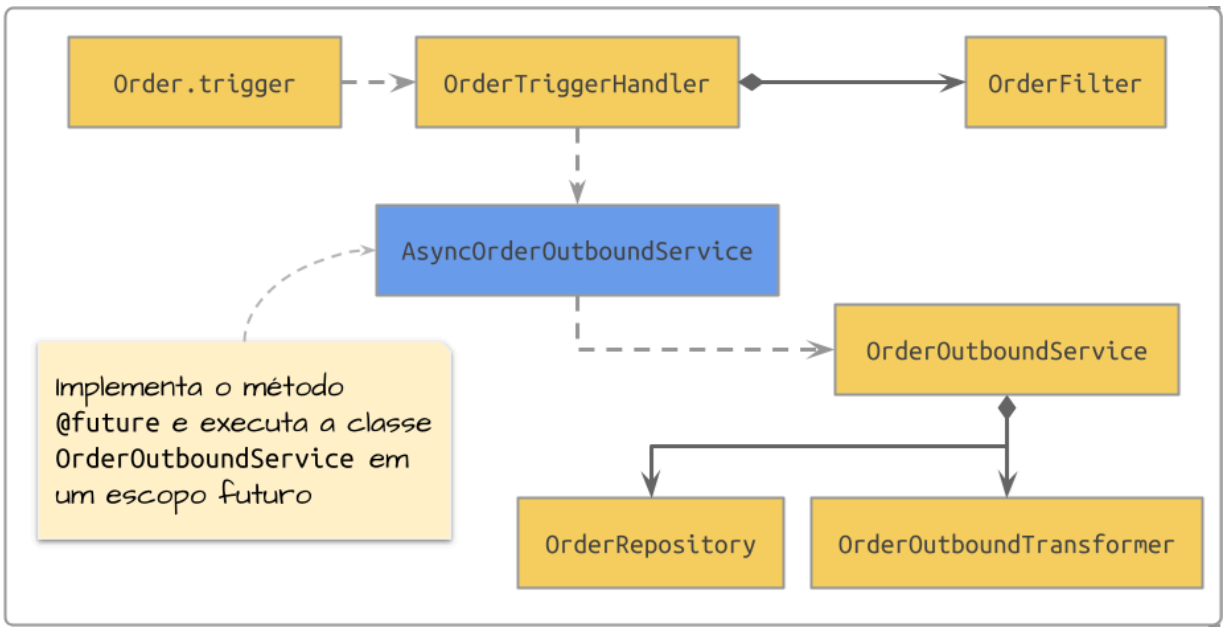


Figura 11.8: Diagrama de classes para o serviço de sincronização de Pedidos

Nesse exemplo vou omitir várias partes de código, pois temos a intenção de apresentar o fluxo de chamada de um *OutboundService* assíncrono. Para ver o código completo, acesse o repositório (https://github.com/ercarval/salesforce-book-snippets).

```
public class OrderOutboundService {

    OrderOutboundTransformer transformer;

    public void sendOrder ( String orderId ) {

        Order order = repository.findById ( orderId );

        OrderOutboundResponse response = send ( order );

        transformer.toOrder ( response , order);

        repository.save ( order );

    }

    private OrderOutboundResponse send ( Order order ) {
```

```
        OrderOutboundRequest request =
                transformer.toOrderOutboundRequest ( order );

        return send ( request );
    }

}
```

Um ponto importante ao utilizar `@future` para chamadas de serviços externos: é necessário adicionar a propriedade `callout=true` na anotação.

```
public class AsyncOrderOutboundService {

    public void send ( List<Order> orders ) {

        for (Order order : orders) {
            send( order.Id );
        }

    }

    @future (callout=true)
    public static void send ( String orderId ) {

        OrderOutboundService service
                                = new OrderOutboundService();

        service.sendOrder ( orderId );

    }

}
```

Agora vamos chamar a classe `AsyncOrderOutboundService` em nossa `OrderTriggerHandler` para todos os *Pedidos* aprovados:

```
public class OrderTriggerHandler extends TriggerHandler {
```

```
    AsyncOrderOutboundService outboundService;

    // construtores e inicialização omitidos

    override
    public void afterUpdate () {

        sendOrderOutbound();

    }

    void sendOrderOutbound () {

        List<Order> approvedOrders =
                        this.filter.byChangedStatus ( newOrders
                                                    , oldOrders
                                                    ,'Approved');

        outboundService.send ( approvedOrders );

    }

}
```

Agora temos um cenário muito mais complexo e próximo da realidade. Ele apresenta um dos principais casos de utilização de chamada futura.

Limites

A plataforma sempre nos delimitará e nos imporá limites. No caso, podemos criar somente 50 chamadas futuras em uma única transação, e esse é um aspecto muito importante quando tratamos de *Trigger*, pois sempre trabalhamos com coleções e podemos ter mais de 50 registros. Com isso, o código apresentado anteriormente pode ultrapassar o limite de chamadas futuras.

Para resolver esse possível problema de limites, o ideal é quebrar em blocos de chamadas futuras, ou seja, o método que realiza a

chamada futura não recebe somente uma `String` , mas uma `List<String>` .

11.4 Utilizando a interface Queueable para chamadas assíncronas

Uma alternativa à chamada futura é a utilização da interface `Queueable` , que, diferentemente da `@future` , permite que sejam passados objetos não primitivos e também permite o encadeamento de chamadas. Vamos alterar a classe `AsyncOrderOutboundService` para utilizar `Queueable` :

```
public class AsyncOrderOutboundService implements Queueable
                                    , Database.AllowsCallouts {

    List<Order> approvedOrders;

    OrderOutboundService service;

    public AsyncOrderOutboundService () {
        service = new OrderOutboundService();
    }

    public AsyncOrderOutboundService (
                                List<Order> approvedOrders) {

        this();
        this.approvedOrders = approvedOrders;

    }

    public void execute (QueueableContext context) {
        send ( approvedOrders );
    }

    public void send ( List<Order> orders ) {
```

```
        for (Order order : orders) {
            service.send( order.Id );
        }

    }

}
```

Note que no código anterior tivemos que utilizar a interface `Database.AllowsCallouts` juntamente à `Queueable`, isso porque a `Queueable` nos permite realizar chamadas assíncronas, porém não permite a realização de *Callouts* de forma automática. Então é necessário utilizar a interface de marcação `Database.AllowsCallouts`, que permite a realização dessas chamadas.

Para consumir uma classe que implementa a interface `Queueable`, é necessário utilizar o método `enqueueJob` da classe `System`, passando como parâmetro a classe `Queueable` que será executada. Dessa forma, a plataforma vai executar a classe de forma assíncrona. Veja no código a seguir o que muda no consumo de uma classe `Queueable`:

```
public class OrderTriggerHandler extends TriggerHandler {

    // construtores e inicialização omitidos

    override
    public void afterUpdate () {

        sendOrderOutbound();

    }

    void sendOrderOutbound () {

        List<Order> approvedOrders =
                    this.filter.byChangedStatus ( newOrders
                                                , oldOrders
```

```
                                                ,'Approved');

        System.enqueueJob (
             new AsyncOrderOutboundService ( approvedOrders ));

    }

}
```

Essa implementação permite que sejam utilizados mais elementos da Orientação a Objetos, pois podemos passar qualquer objeto como parâmetro para a instância da classe `Queueable`, o que deixa o código mais legível e elegante. Vamos testar nossa nova implementação de `AsyncOrderOutboundService`. Observe na classe a seguir que basicamente alteramos a chamada da classe:

```
@isTest
public class AsyncOrderOutboundServiceTest {

    @isTest
    public static void shouldSendOrder() {

        String reponsePayload =
                            '{"orderNumber":"12323212132"}';

        Map<String, SObject> records =
                OrderFixtureFactory.createIntegratedScenario();

        Order order = (Order) records.get('order');

        Test.setMock ( HttpCalloutMock.class
            , new OrderOutboundServiceMock ( reponsePayload ));

        Test.startTest();

        List<Order> orders = new List<Order> {order};

        System.enqueueJob (
             new AsyncOrderOutboundService ( orders ));
```

```
        Test.stopTest();

        Order storedOrders = [ SELECT PoNumber
                               FROM Order LIMIT 1];

        System.assertEquals( '12323212132'
                           , storedOrders.PoNumber );

    }

}
```

Todos os fluxos assíncronos são processados após a execução do comando `Test.stopTest()`. Dessa forma, é possível executar os fluxos assíncronos em classes de teste.

Note que, nesse último exemplo, aplicamos vários pontos estudados anteriormente, como a utilização de `HttpCalloutMock` e a criação de cenários de testes complexos através do `OrderFixtureFactory.createIntegratedScenario`, que é o responsável por criar todas as dependências para criação de um *Pedido* (`Order`).

#se-liga-na-info 😎

Você pode estar se perguntando: qual implementação devo utilizar para realizar chamadas assíncronas, `@future` ou `Queueable`? Na verdade, não há um certo ou errado, mas, sim, um contexto.

Pode ser mais simples de controlar a quantidade de chamadas assíncronas utilizando `@future`, assim como pode ser mais simples de trabalhar com listas prontas de objetos, como `List<Order>`, utilizando `Queueable`. O importante é ter as duas opções em mente e escolher a melhor opção para seu projeto.

Este capítulo trouxe duas alternativas para tralharmos com chamadas assíncronas. Elas são utilizadas para trabalhar com fluxos transacionais, ou seja, não são recomendadas para alto volume de dados. Para isso, a Salesforce fornece outros caminhos que também são assíncronos, mas com propósitos diferentes. Você também teve a oportunidade de unificar diversas técnicas de desenvolvimento através de códigos pequenos e elegantes.

Capítulo 12
Trabalhe com classes agendadas

Em muitos cenários de negócio, necessitamos trabalhar com agendamento de tarefas, como: fechamento de horas do mês, apuração de resultados de um determinado período, envio de e-mails. E como fazemos isso através da plataforma Salesforce? Implementando a interface `Schedulable`, que permite que classes APEX sejam executadas em um determinado horário.

Considere o cenário de nossas integrações anteriores, onde utilizamos um objeto de log para armazenar informações sobre a execução da integração. Esse objeto tem a tendência de crescer, gerando um custo de armazenamento desnecessário. Para resolver isso, devemos manter um nível de retenção de logs, removendo registros antigos. Esse é um cenário perfeito para utilização de **classes agendadas**, pois podemos executar uma classe em um horário predeterminado.

Vamos começar pela protagonista, a `PurgeLogJob`, que deve implementar a interface `Schedulable`, que determina o comportamento de ser "agendável". Essa interface exige que o método `execute` seja implementado, e é ele que será acionado pelo agendamento. Veja a implementação a seguir:

```
public class PurgeLogJob implements Schedulable {

    LogRepository repository;

    public PurgeLogJob () {
        repository = new LogRepository();
    }

    public void execute (SchedulableContext context) {

        List<Log__c> logs =
```

```
            repository.findByCreatedLessThanDays (10);

        repository.remove (logs);

    }

}
```

Bem simples! Basta implementar o método `execute` para termos uma classe que pode ser agendada! Em nossa implementação utilizamos a `LogRepository` para determinar os registros que deverão excluídos e informamos a quantidades de dias que deverão ser considerados para retenção. Veja o código a seguir, onde é utilizada uma *literal de data* para realizar esse cálculo:

```
public class LogRepository extends AbstractRepository {

    public List<Log__c> findByCreatedLessThanDays (
                                            Integer days ) {

        return [SELECT Id
                FROM Log__c
                WHERE CreatedDate < LAST_N_DAYS:days ];

    }

}
```

#se-liga-na-info 😎

As literais de data ajudam bastante no dia a dia, pois não temos a necessidade de fazer cálculos com datas para realizar uma pesquisa. Para ver a lista completa dessas literais e outros recursos voltados para data, acesse a documentação de data, em: https://developer.salesforce.com/docs/atlas.en-us.soql_sosl.meta/soql_sosl/sforce_api_calls_soql_select_dateformats.htm.

Agora que você viu como é simples criar uma classe que pode ser agendada, vamos ver como realizamos o agendamento propriamente dito. Para isso, vamos utilizar o método `schedule` da classe `System`. Esse método recebe como parâmetro o nome do *job* (que é exclusivo), uma expressão *CRON* e uma instância de `Schedulable`.

> A expressão *CRON* é uma expressão que permite definir o horário de execução da tarefa, muito comum em sistemas operacionais baseado em UNIX.

Veja o exemplo a seguir, que executará a nossa classe todos os dias às `03:00:00`. Não se preocupe com a expressão *CRON*, ela é a parte mais difícil desta implementação:

```
String jobId = System.schedule ( 'Purge Logs'
                                ,'0 0 3 * * ?'
                                , new PurgeLogJob());
```

Simples, não? Como comentei, a expressão CRON é a parte mais difícil, então vamos vê-la com mais detalhes.

12.1 Entendendo a expressão CRON

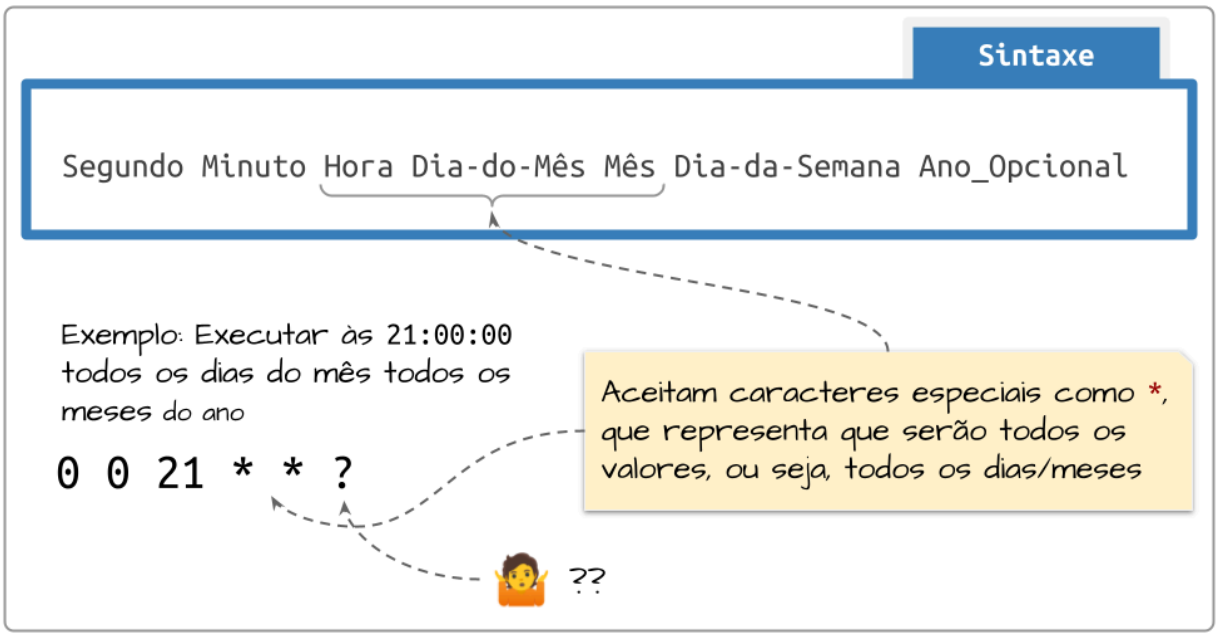


Figura 12.2: Sintaxe da expressão CRON

Vamos ver cada elemento da expressão CRON:

- **Segundo:** valor obrigatório entre 0 e 59, representando o segundo do minuto;
- **Minuto:** valor obrigatório entre 0 e 59, representando o minuto do horário;
- **Hora:** valor entre 0 e 23, representando o horário. Aceita os caracteres especiais `, - * /`;
- **Dia do mês:** valor obrigatório entre 1 e 31, representando o dia do mês. Aceita os caracteres especiais `, - * ? / L W`;
- **Mês:** valor obrigatório entre 1 e 12 ou utilizando as literais (`JAN`, `FEB`, `MAR`, `APR`, `MAY`, `JUN`, `JUL`, `AUG`, `SEP`, `OCT`, `NOV`, `DEC`), representando o mês do ano. Aceita os caracteres especiais `, - * /`;
- **Dia da semana:** valor obrigatório entre 1 e 7 ou utilizando as literais (`SUN`, `MON`, `TUE`, `WED`, `THU`, `FRI`, `SAT`), representando o dia da semana. Aceita os caracteres especiais `, - * ? / L #`;
- **Ano**: valor opcional, representando o ano. Aceita `null` ou um valor entre 1970 e 2099.

Para entender os caracteres especiais, vejamos os mais utilizados:

Caractere especial	Descrição
`,`	Delimitador de valores utilizado para apresentar uma sequência de valores, como `FEB, MAY, OCT` ou `1,7,16,21`
`-`	Especifica um intervalo. Por exemplo, `MON-FRI` ou `2-11`
`*`	Especifica todos os valores. Por exemplo, executar todos os dias do mês, se dia estiver utilizando `*`
`?`	Determina que não há um valor específico. Deve ser aplicado somente aos elementos `Dia do Mês` ou `Dia da Semana`. Quando expressar o `Dia do Mês`, deve-se utilizar esse caractere no `Dia da Semana`, ou, quando expressar o `Dia da Semana`, ele deve ser utilizado no `Dia do Mês`.
`/`	Define incrementos. O primeiro valor representa o início do intervalo, e o último determina o valor do incremento. Por exemplo, se expressar `1/4` no mês, o agendamento executará a cada quatro meses, iniciando a partir do primeiro mês.

Vamos ver alguns exemplos:

- `0 15 * ? * MON-FRI` : executará no minuto 15 de todas as horas de segunda à sexta. Como foi informado o dia da semana, foi utilizado o caractere `?` para representar que não há um valor específico em dia do mês;
- `0 0 3 * * ?` : executará todos os dias às 3:00:00 da manhã;
- `0 0 1,3,21,23 * * ?` : executará todos os dias à 1 hora da manhã (1h00), às 3 da manhã (3h00) e, depois, às 21 horas (21h00) e às 23 horas (23h00) da noite.

Espero ter ajudado, mas não se preocupe em decorar, eu mesmo nunca consigo :). Sempre recorro à documentação para criar o agendamento.

12.2 Rastreando os agendamentos

Quando criamos um agendamento, a plataforma retorna o `Id` correspondente e, com isso, é possível obter informações sobre o seu status, ou a próxima execução, entre outras informações. Os objetos responsáveis por armazenar essas informações são `CronTrigger` e `CronJobDetail`:

```
public List<CronTrigger> findByName ( String name ) {

    return [ SELECT Id , CronJobDetail.Name,
                TimesTriggered, NextFireTime
             FROM CronTrigger
             WHERE
             CronJobDetail.Name = :name ];
}
```

Através desses objetos podemos rastrear o status de nosso agendamento, a próxima execução ou quantas vezes foi executado.

Agora que você conhece o objeto `CronTrigger`, você pode imaginar que, para cancelar um agendamento, basta remover do objeto, mas, na verdade, não. Para abortar/cancelar um agendamento, é necessário utilizar o método `abortJob` da classe `System`, informando o `Id` do agendamento:

```
CronTriggerRepository repository = new CronTriggerRepository();

CronTrigger job = repository.findByName('PurgeLogJob');

System.abortJob ( job.Id );
```

#e-agora-como-eu-testo 🤷

12.3 Como testar o agendamento?

As classes que implementam a interface `Schedulable` são consideradas assíncronas, portanto os métodos de teste devem utilizar o delimitador `Test.startTest()` e `Test.stopTest()` para que o *job* seja executado. Veja o código a seguir, que realiza o teste de nossa classe de agendamento:

```
@isTest
static void givenTimeReachedWhenScheduleThenExecutePurgeLog () {

    String cronExpression = '0 0 0 * * ?';

    DateTime oldDateLog = DateTime.now().addDays(-11);

    Log__c logToPurge = LogFixtureFactory.createLog();

    Test.setCreatedDate( logToPurge.Id , oldDateLog );

    Log__c logToKeep = LogFixtureFactory.createLog();

    Test.startTest();

    System.schedule ( 'PurgeJobTest'
                    , cronExpression
                    , new PurgeLogJob () );

    Test.stopTest();

    List<Log__c> logs = [ SELECT Id FROM Log__c  ];
```

```
    System.assertEquals ( 1, logs.size() );

    System.assertEquals ( logToKeep.Id , logs.get(0).Id );

}
```

Essa classe de teste trouxe mais um recurso da plataforma, o método `setCreatedDate` da classe `Test`, que tem o propósito de redeterminar a data de criação de um registro em um escopo de testes. Através desse recurso, foi possível criar um registro com uma data antiga, simulando o comportamento de um registro que foi criado antes do agendamento. Esse recurso é muito útil em cenários onde precisamos criar uma massa de dados com um determinado intervalo de datas.

Ela também criou o agendamento utilizando a expressão CRON `'0 0 0 * * ?';`, que não é a data desejada, porém o objetivo desse teste não é avaliar a data do agendamento ou a expressão CRON, mas, sim, a sua execução. Lembrando que a classe agendada será acionada logo após o comando `Test.stopTest();`, onde processará o método `execute` da classe `PurgeLogJob`.

#fica-a-dica 🤔

Durante os ciclos de teste, o objeto `CronTrigger` mantém os registros existentes na *Org*. Dessa forma, os métodos de teste deverão alterar o nome do agendamento, para não gerar um erro de nome duplicado.

Atenção aos limites

Todos os recursos da plataforma são gerenciados. Não podemos criar agendamentos de forma ilimitada, pois podemos ter somente `100` agendamentos. Esse limite é compartilhado com o de processamento assíncrono, que tem a capacidade de execução de

no máximo de `250.000` por dia ou o número de licenças de usuário em sua *Org* multiplicado por `200`, aplicando o que for maior.

Capítulo 13
Trabalhando com alto volume de dados

Durante todo o decorrer do livro, eu mencionei sobre os limites da plataforma, e com isso você pode acabar pensando que ela é limitada. Essa ideia não está errada, mas também não está certa :). Todo sistema tem limites impostos pelo hardware, pela linguagem, pela arquitetura, entre outros aspectos. Eles não possuem um monitor explícito como a Salesforce e, com isso, é possível fazer uma verdadeira "lambança" no código, como:

- Realizar operações de DML/SQL dentro de loops;
- Obter até 50 milhões de registros de uma única vez;
- Tentar processar milhares de registros sincronicamente.

Com isso em mente, os limites impostos pela Salesforce vêm para nos lembrar de que não podemos fazer tudo o que desejamos sem pensar na arquitetura, na volumetria de dados e no tipo de resposta que deve ser dada ao usuário. Por isso é importante ter em mãos uma caixa de ferramentas, com opções para prover a melhor solução para a necessidade do usuário. Neste capítulo, vou acrescentar mais uma dessas ferramentas: a interface `Batchable`. Ela traz a capacidade de processar alto volume de dados dentro da plataforma, porém sempre respeitando os limites.

13.1 Interface Batchable

As classes que implementam a interface `Batchable` são comumente chamadas de *Batch*. Elas têm a finalidade de processar grandes lotes de dados, sendo esta uma de suas principais características: ela não processa os dados de uma única vez, mas fragmenta os dados de forma que sempre tenha um bloco pequeno para processar. Com

isso, garantimos que em cada ciclo de execução não rompamos os limites impostos pela Salesforce. Essa interface necessita que três métodos sejam implementados:

- **`start`:** responsável por iniciar o *Batch* (a classe), provendo uma instância de `Database.QueryLocator` ou `Iterator<T>`. Ele retorna uma `query` que tem a capacidade de ser separada em lotes de execução de forma automática;
- **`execute`:** responsável pela execução propriamente dita. Recebe dois parâmetros: `Database.BatchableContext`, que possui informações como o `Id` do *Batch* gerado; e `List<SObject>`, que corresponde a uma parte do resultado da *query* executada no bloco `start`;
- **`finish`:** responsável por finalizar o *Batch*.

Com essa estrutura provida pela interface `Batchable`, temos a capacidade de processar um alto volume de dados sempre respeitando os limites (até 50 milhões de registros), pois ela fragmenta a consulta inicial em blocos, delegando o processamento para o método `execute`.

Agora vamos aplicar essa nova ferramenta em nosso cenário de limpeza de logs. Vamos considerar que não prevemos que poderíamos ter muitos logs antigos e que, com isso, poderíamos atingir os limites da plataforma. Para corrigir esse problema, vamos utilizar a interface `Batchable`:

```
public class PurgeLogsBatch implements
                                Database.Batchable<SObject> {

    LogRepository repository;

    Integer daysToKeep;

    public PurgeLogsBatch () {
        this.repository = new LogRepository();
        this.daysToKeep = 30;
    }
```

```
    public Database.QueryLocator start(
                            Database.BatchableContext context) {

        Integer days = this.daysToKeep;

        return Database.getQueryLocator('SELECT Id  FROM Log__c WHERE
CreatedDate < LAST_N_DAYS:days');

    }

    public void execute ( Database.BatchableContext context
                        , List<Log__c> logs) {

        repository.remove (logs);

    }

    public void finish ( Database.BatchableContext context ) {

    }

    public static void start () {
        Database.executeBatch ( new PurgeLogsBatch() );
    }

}
```

Ao ver o código anterior, note que especializamos a interface `Database.Batchable` com `SObject`. Dessa forma, é esperado que o método `start` de nossa classe retorne um `Database.QueryLocator`, ou seja, sempre que você desejar utilizar uma *query* simples como entrada para seu *Batch*, a interface deverá estar tipada com `SObject`.

Mas se eu especializar a interface `Database.Batchable` com outro tipo, o que acontece? O compilador do APEX entende que sua classe deve retornar um `Iterator<T>` como tipo de retorno no método `start`. Isso porque a interface é utilizada para retornos complexos, que não

são uma *query* simples, por exemplo, quando desejo criar um *Batch* para uma classe customizada ou um conjunto de consultas. É um cenário bem específico e deve-se avaliar se não pode ser tratado com o `Database.QueryLocator`, já que adiciona uma complexidade considerável à classe de *Batch*.

Voltando ao nosso código anterior, o método `start` está utilizando o `Database.getQueryLocator`, que pode receber o SOQL representado por uma `String` ou por uma *query* `[SELECT .. FROM .. ]`.

Para executar o *Batch*, utilizamos o método `executeBatch` da classe `Database`. Ele tem duas assinaturas:

- **`executeBatch (Batchable)`:** executa a classe concreta passada como parâmetro, quebrando o conteúdo produzido pelo método `start (QueryLocator)` em 200 registros.
- **`executeBatch (Batchable, Integer)`:** executa a classe concreta passada como parâmetro, quebrando o conteúdo em blocos definidos pelo parâmetro `Integer` passado.

Observe a imagem a seguir, onde é apresentado o fluxo de execução da `Batchable` e o fluxo de execução em blocos da `Batchable`:

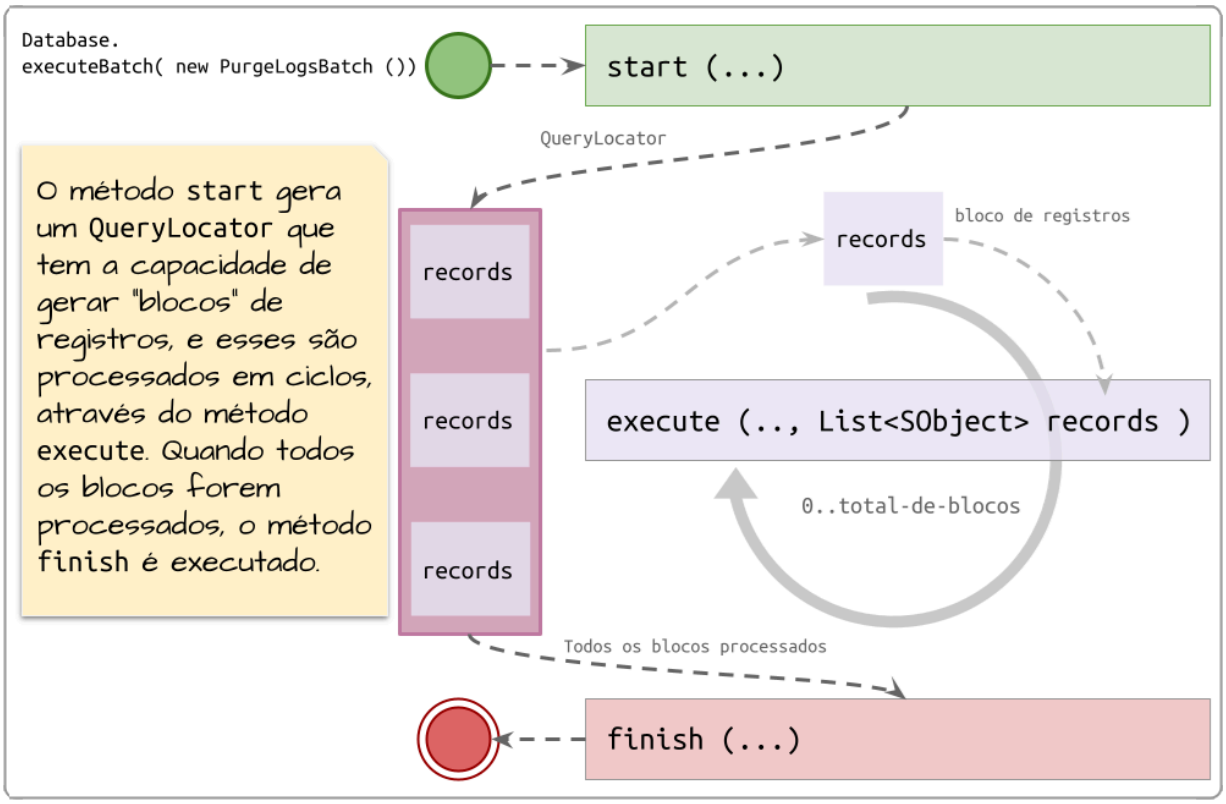


Figura 13.1: Processamento de Batches

Em todos os métodos recebemos uma instância de `Database.BatchContext`, que nos permite rastrear o *Batch* atual através do método `getJobId`.

Rastreie o processamento do *Batch*

Quando tratamos de processos assíncronos, não temos o resultado do processamento imediatamente, e cada implementação fornece formas de acompanhar o seu processamento. No caso de um *Batch*, a Salesforce utiliza o objeto `AsyncApexJob`, com o qual temos a capacidade de identificar o status, o total de lotes do *Batch* (`TotalJobItems`) e de lotes processados (`JobItemsProcessed`).

Podemos acompanhá-los também através da tela de **Trabalhos do Apex**, que pode ser acessada através do menu *Configurações* > *Ambientes* > *Trabalhos* > ***Trabalhos do Apex***.

Figura 13.2: Tracking

Testando classes `Batchable`

Testes voltados para classes *Batch* seguem o mesmo princípio dos testes de classes assíncronas. Sua execução ocorrerá logo após a chamada do método `Test.stopTest()`. Então, basicamente, precisamos fazer o `setup` dos dados e executar a classe *Batch* através do `Database.executeBatch`. Bem simples!

Gosto de destacar que esses testes devem ser voltados para *estrutura*, não para *volumetria*. Então, você não precisa criar diversos registros para realizar o teste; dessa forma, o `setup` desse tipo de teste fica mais rápido e mais eficiente.

```
@TestSetup
static void setup () {

    DateTime oldDateLog = DateTime.now().addDays(-31);

    Log__c logToPurge = LogFixtureFactory.createLog();

    Test.setCreatedDate( logToPurge.Id , oldDateLog );

    Log__c logToKeep = LogFixtureFactory.createLog();

}

@isTest
static void givenOldLogsWhenRunPurgeJobThenKeepOnlyLastMonth(){
```

```
        Test.startTest();

        Database.executeBatch( new PurgeLogsBatch() );

        Test.stopTest();

        Integer totalOfLogs = Database.countQuery (
                                'SELECT COUNT(Id) FROM Log__c');

        System.assertEquals ( 1, totalOfLogs );

}
```

Nesse método de teste eu trouxe um formato um pouco diferente dos demais: a utilização da anotação `@TestSetup`. Ela faz com que o método anotado seja executado antes de todos os métodos anotados com `@isTest`. Essa abordagem é válida quando **TODOS** os métodos dependem do mesmo dado. Dessa forma, temos um benefício de não repetir o processo de `setup` em todos os métodos.

> **#se-liga-na-info** 😎
>
> Caso você tenha cenários de teste que não necessitem desse `setup`, não recomendo que ele seja utilizado, pois aumenta consideravelmente o tempo de execução dos testes. Dê preferência pelo `setup` local em cada método. Faço esse alerta, pois há quem utilize essa abordagem de forma equivocada sem se preocupar com o tempo gasto em classes de teste. Vale lembrar que, para cada método de teste, os dados serão criados no início e apagados no final de sua execução.

Assim como outras classes de teste de processos assíncronos, devemos envolver a execução do *Batch* com os comandos `Test.startTest()` e `Test.stopTest()`. Isso fará com que o *Batch* seja acionado logo após a execução do `stopTest()`.

Batches e *callouts*

Da mesma forma que interfaces `Queueable` não permitem a realização de *callouts*, classes que implementam `Database.Batchable` necessitam implementar a interface de marcação `Database.AllowsCallouts` para que possam ser realizadas as chamadas externas.

Batches e chamadas futuras

Classes do tipo *Batch* são classificadas como assíncronas, com isso não é possível realizar chamadas futuras (`@future`) dentro de um escopo de *Batch*. Porém podemos utilizar classes que implementam a interface `Queueable` , iniciando um fluxo assíncrono dentro de outro.

Esse é um cenário atípico, e recomendo que analise todas as possibilidades de utilização, pois chamadas futuras ou `Queueable` são geralmente aplicadas a fluxos transacionais e podem causar estouro de limites.

13.2 Encadeamento de Batches

A principal característica de um *Batch* é a capacidade de trabalhar com blocos de processamento, mas para um único objeto. Quando temos mais de um objeto com alto volume de dados, utilizamos encadeamento de *Batches*. Essa é uma prática muito utilizada, que consiste em criar um novo *Batch* no método `finish` da classe `Batchable` . Dessa forma, ao finalizar o *Batch* atual, um novo *Batch* será criado.

Agora imagine o cenário onde precisamos fazer a troca de proprietário das *Contas*, com base na nova estrutura de *Executivos*. Os proprietários são determinados com base no *Agrupamento* de

Contas para cada *Executivo*. Veja o relacionamento dos objetos a seguir:

Figura 13.4: Relação de hierarquia entre Executivo de Contas e Conta

Diariamente, o sistema de nosso *Cliente* realiza uma carga no objeto *Grupo de Executivo de Conta* `AccountManagerGroup__c` redistribuindo as *Contas* entre os *Executivos*. Com isso precisamos realizar a troca de proprietário das *Contas* para refletir essa nova disposição da carteira de clientes de cada *Executivo*.

Como você já viu anteriormente, a *Conta*, que representa o *Cliente*, é o centro de um CRM. Com isso em mente, a carteira de um Executivo deve ter uma visão completa de tudo que está relacionado à *Conta*. Geralmente definimos o *Executivo* como o proprietário de diversos registros em objetos referentes à *Conta*, como:

- Oportunidade;
- Caso;
- Contato;
- Contrato.

Essa lista pode ficar enorme, e depende muito da necessidade do CRM implantado. Nesse exemplo vamos nos limitar somente a *Oportunidade* e *Caso*.

Como temos diversos objetos para atualizar, é necessário fazer o levantamento de volumetria de registros por objeto em relação à *Conta*, levando em consideração o pior cenário em quantidade de registros. Com base em nosso levantamento, podemos ter:

- Até 200 *Oportunidades* abertas por *Conta*;
- Até 200 *Casos* abertos por *Conta*.

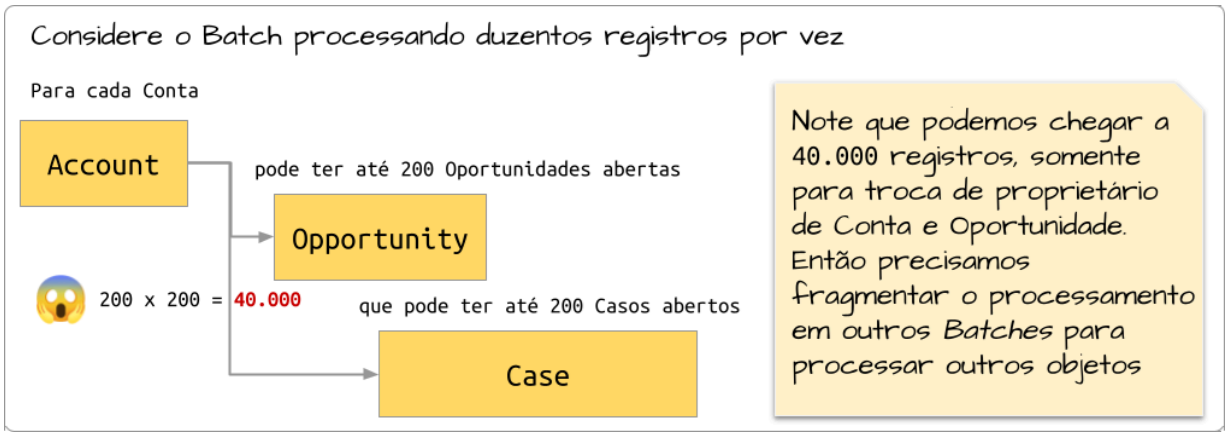


Figura 13.5: Hierarquia de objetos relacionados a Conta

Considerando que vamos fragmentar o *Batch* em blocos de 200 registros, podemos fazer a atualização de *Conta* e *Oportunidade* simultaneamente, com uma folga de 10 mil registros para atingir o limite. Porém, não podemos realizar a troca do proprietário do *Caso* simultaneamente, sendo necessário processar em outro *Batch*. Veja a implementação da classe `ChangeAccountAndOpportunityOwnerBatch` :

```
public without sharing class ChangeAccountAndOpportunityOwnerBatch
                        implements Database.Batchable<SObject> {

    AccountManagerGroupRepository repository;

    OpportunityRepository opportunityRepository;

    public ChangeAccountAndOpportunityOwnerBatch () {

        repository = new AccountManagerGroupRepository();
        opportunityRepository = new OpportunityRepository();

    }

    public Database.QueryLocator start (
                            Database.BatchableContext context) {
```

```
    Date today = Date.today();

    Time startDay = Time.newInstance(0,0,0,0);

    Time endDay = Time.newInstance(23,59,59,999);

    DateTime startDate = Datetime.newInstance ( today
                                                , startDay );

    DateTime endDate = Datetime.newInstance ( today
                                              , endDay );

    return repository.findLastModifiedGroups ( startDate
                                              , endDate);

}

public void execute ( Database.BatchableContext context
            , List<AccountManagerGroup__c> managerGroups ) {

    List<Account> accounts;

    accounts = buildChangedAccountOwner (managerGroups);

    List<Opportunity> opportunities;

    opportunities = buildChangedOpenOpportunities (accounts);

    repository.save(accounts);

    opportunityRepository.save(opportunities);

}

List<Account> buildChangedAccountOwner (
               List<AccountManagerGroup__c> managerGroups) {

    List<Account> accounts = new List<Account> ();
```

```
    for (AccountManagerGroup__c managerGroup:managerGroups){

        accounts.add ( buildAccount (managerGroup) );

    }

    return accounts;

}

Account buildAccount (AccountManagerGroup__c managerGroup){

    Account account = new Account ();
    account.Id = managerGroup.Account__c;
    account.OwnerId = managerGroup.AccountManager__r.OwnerId;

    return account;

}

List<Opportunity> buildChangedOpenOpportunities (
                                    List<Account> accounts) {

    Map<Id, Account> indexedAccounts;
    indexedAccounts = new Map<Id, Account> (accounts);

    List<Opportunity> opportunities;

    opportunities = opportunityRepository
                        .findOpenOpportunties (
                                indexedAccounts.keySet());

    if (opportunities.isEmpty()) {
        return new List<Opportunity>();
    }

    for (Opportunity opportunity : opportunities ) {
```

```
            Account account = indexedAccounts.get(
                                        opportunity.AccountId);

            opportunity.OwnerId = account.OwnerId;
        }

        return opportunities;

    }

    public void finish ( Database.BatchableContext context ) {
        Database.executeBatch ( new ChangeCaseOwnerBatch() );
    }

}
```

A classe `ChangeAccountAndOpportunityOwnerBatch` tem a responsabilidade de processar todas as modificações realizadas no *Grupo de Executivo de Contas*, trocando o proprietário das *Contas* e suas *Oportunidades* abertas, mantendo a segurança e a visibilidade dos registros com o *Executivo* correto.

Agora veja a implementação da classe `ChangeCaseOwnerBatch`, que utiliza da mesma fonte que o *Batch* anterior para trocar os proprietários dos *Casos* abertos:

```
public without sharing class ChangeCaseOwnerBatch
                        implements Database.Batchable<SObject> {

    CaseRepository caseRepository;

    AccountManagerGroupRepository repository;

    AccountManagerGroupFilter filter;

    public ChangeCaseOwnerBatch() {
        repository = new AccountManagerGroupRepository();
        caseRepository = new CaseRepository();
        filter = new AccountManagerGroupFilter();
```

```
}

public Database.QueryLocator start (
                        Database.BatchableContext context) {

    Date today = Date.today();

    Time startDay = Time.newInstance(0,0,0,0);

    Time endDay = Time.newInstance(23,59,59,999)

    DateTime startDate = Datetime.newInstance ( today
                                              , startDay );

    DateTime endDate = Datetime.newInstance ( today
                                             , endDay );

    return repository.findLastModifiedGroups ( startDate
                                             , endDate);

}

public void execute ( Database.BatchableContext context
            , List<AccountManagerGroup__c> managerGroups ) {

    List<Case> cases = buildCases (managerGroups);

    caseRepository.save(cases);

}

List<Case> buildCases (
                List<AccountManagerGroup__c> managerGroups) {

    List<Case> cases = findOpenCases (managerGroups);

    if (cases.isEmpty()) return cases;

    Map<String, AccountManagerGroup__c> groups;
```

```
        groups = filter.indexByAccount (managerGroups);

        // A palavra case é reservada, então trocamos por "caze"
        // em vez de chamar de cs!
        for (Case caze : cases ) {

            caze.OwnerId = groups.get(caze.AccountId)
                                    .AccountManager__r.OwnerId;

        }

        return cases;

    }

    List<Case> findOpenCases (
                    List<AccountManagerGroup__c> managerGroups) {

        List<String> accountIds = filter
                            .pullAccountIds ( managerGroups );

        return caseRepository.findOpenCases ( accountIds );

    }

    public void finish ( Database.BatchableContext context ) {
    }

}
```

Nossa implementação ficou bem simples, mas muito próxima do mundo real. Cenários como esse são comuns, e devemos ter a volumetria em mente sempre para a tomada de decisão de como arquitetar nossa solução.

#se-liga-na-info 😎

Classes do tipo *Batch* nascem com um propósito bem definido. Podemos utilizá-la para a implementação do algoritmo de processamento, que foi o caso da classe anterior. Ela delegou somente o acesso ao banco para classes do tipo `Repository`, pois o algoritmo era muito pequeno e conciso, portanto não havia a necessidade de criar outra classe para esse processamento.

A tomada de decisão para criação de outras classes deve ser ligada à complexidade do processamento, de modo a garantir a manutenibilidade do *Batch*.

Agora você tem mais uma ferramenta no seu kit. Os exemplos apresentados são muito próximos do real, e posso dizer o mesmo da base de código. Então aplique já esse conhecimento, para não esquecer!

Mas você acha que acabou por aqui? Claro que não! O próximo capítulo vai aplicar tudo o que vimos de Orientação a Objetos para melhorar nossa prática de testes.

Parte 5: Seja um(a) ninja nos testes

> "*Você só não aprende aquilo que você desiste de aprender!*"
>
> — Autor desconhecido

Nos capítulos anteriores, você viu algumas práticas de teste que ajudaram você na sua jornada de se tornar um(a) ninja, mas será que você tem consciência de todos os desafios que terá de enfrentar?

Vou ajudar com uma listinha:

- Manter os testes limpos;
- Fazer com que eles cumpram o seu propósito;
- Fazer com que eles sejam fáceis de entender, ou seja, legíveis.

Vários desses pontos foram abordados anteriormente, quando vimos padrões de nomes para métodos de teste. Também vimos como organizar os dados de teste através de `FixtureFactories`.

Aí vem a pergunta: "_Então já posso me considerar um(a) ninja, Bisso?".

Posso dizer que você está no caminho certo, mas ainda faltam alguns *Jutsus* para você aprender. Imagine uma Org Salesforce com mais de 1.500 testes, e todos eles fazendo operações de criação/modificação de registros. Possivelmente a execução de todos esses testes levaria quase uma hora ou mais.

Viu agora o tamanho do buraco? Esse é um cenário real de muitos clientes que operam com Salesforce. Mas como sair dele? Este é nosso desafio: tornar nossos testes mais **limpos**, **legíveis** e **rápidos** de serem executados.

Para começar, vamos ver os princípios que norteiam uma boa classe de teste e, em seguida, técnicas que farão com que o jogo vire! Então "bora" desvendar esses novos *Jutsus*!

Capítulo 14
F.I.R.S.T.

Na introdução desta parte do livro, apresentei alguns pontos que indicam que nossos testes estão bons, mas gosto de dizer que devemos utilizar os princípios como um balizador. Quando somos norteados por princípios, temos mais clareza e propósito para um determinado comportamento, e isso não é diferente com software.

Os princípios descritos por Robert C. Martin em seu livro *Clean Code* nos trazem mais clareza do que de fato representa um teste limpo. Esses princípios e regras são representados pelo acrônimo em inglês **F.I.R.S.T.**:

- ***Fast*:** um teste deve ser **rápido**, deve retornar um feedback o mais rápido possível, pois, em caso de lentidão, existe uma tendência de que ele deixe de ser executado. Ninguém quer esperar 10 minutos para uma resposta de um teste ou um conjunto de testes;
- ***Isolated/Independent*:** um teste deve ser **isolado e independente**, não deve depender de outro para que possa ser executado. Devemos poder executar cada teste de forma independente, na ordem em que desejarmos;
- ***Repeatable*:** um teste deve ser **repetível**, deve poder ser executado em qualquer ambiente e deve ter um resultado determinístico. Não podemos ter um teste que ora dá um resultado, ora outro. Por isso, **não utilize** `@isTest(seeAllData=true)` , pois utiliza fonte de dados variáveis e, portanto, quebra esse princípio.

- ***Self-Verifying*:** um teste deve ser **autoverificável**, e sempre garantir um resultado verdadeiro ou falso. Ou seja, não podemos simplesmente fazer um teste com a intenção de somente executar o código produtivo, **devemos nos utilizar de asserções** com a intenção de garantir que ele cumpriu o seu propósito.
- ***Timely*** (**pontual**): os testes devem ser escritos em tempo hábil, de preferência imediatamente antes da escrita do código. Caso demore para ser escrito, poderá ser difícil de cobrir parte do código produtivo, fazendo com o que o teste fique ineficiente. Esse princípio também apresenta que devemos criar nosso código produtivo pensando como podemos testá-lo melhor.

`#fica-dica-clean-code`

A prática apresentada no princípio ***Timely*** é chamada de *Test First*. Ela vem do TDD, *Test-Driven Development*, onde escrevemos o código de teste antes do código produtivo. Essa prática pode trazer resultados positivos para o código que será criado. Ela produz uma visão mais empática, uma vez que você passa a construir com o olhar de consumo, podendo alterar a forma como será implementado o código produtivo.

A prática de *Test First* é um desafio de disciplina, pois inicialmente parece contraintuitivo criar o teste antes do código produtivo. Ela exige auto-organização para que se torne um hábito.

Em todas as classes de testes apresentadas até então, aplicamos, de certa forma, os princípios F.I.R.S.T. Por exemplo, quando utilizamos classes como `FixtureFactory` para criar dados de testes, estamos tratando os princípios *Independent* e *Repeatable*. Quando definimos

asserções em todos os testes, estamos tratando o princípio *Self-Verifying*.

Mas o princípio **Fast** foi o único que, de fato, não foi tratado explicitamente. Vamos reavaliar a prática de testes que aplicamos até então.

Na maioria dos métodos de teste nós utilizamos as classes `FixtureFactory` para criar os dados de teste para a cobertura de nossos códigos. Vale destacar que, a cada método de teste onde realizamos uma operação *DML*, é criado um fluxo transacional, que será descartado após a execução do método.

> **#se-liga-na-info** 😎
>
> Esse tipo de teste é chamado de **teste integrado**, pois dependemos de uma fonte de dados externa para realizá-los.

Os *testes integrados* têm um alto custo de processamento e tempo, principalmente quando temos milhares de testes para realizar. A execução de todos os testes impacta diretamente em tempo de implantação (*deployment*), que também está diretamente ligado ao custo de operação, tanto para seu cliente quanto para a empresa que o emprega. Isso porque, geralmente, implantações em produção ocorrem fora do horário comercial.

Com essa volumetria de testes, não basta otimizar os `FixtureFactories` . Isso até poderia reduzir um pouco o tempo de *deployment*, mas nada significativo, ou seja, o tempo não diminuiria drasticamente!

Para superar esse desafio, precisamos ampliar nosso conhecimento de práticas de teste, sempre balizadas pelos princípios apresentados. No próximo capítulo, veremos como utilizar as práticas descritas por

Gerard Meszaros no livro *XUnit Test Patterns* para nos apoiar nessa nossa missão de deixar nossos testes *limpos* e *rápidos*.

Capítulo 15
Mocks

Vimos que através de testes integrados não conseguimos atingir o objetivo de seguir os princípios do **F.I.R.S.T.**. O custo de operação com o banco, inserção, atualização, remoção e, depois, o *rollback* de tudo, é muito alto. Para atingir todos os princípios, precisamos adicionar mais um item à nossa caixa de ferramentas: os ***Mocks***. Vamos adicionar os *Mocks* com o intuito de produzir testes **menos dependentes e mais rápidos**.

O termo ***Mock*** já foi abordado no capítulo 9, nos testes de classes *Callout*, mas agora vamos explorar todas as variabilidades sobre esse tema, pois ele é uma generalização de um conceito que veio diretamente do Cinema para o código, que são os **dublês** de teste.

Você provavelmente já tem uma ideia do que é um dublê no contexto do Cinema: são profissionais utilizados para substituir um determinado ator/atriz em uma cena perigosa. Esse conceito foi portado para o universo de software para podermos testar códigos sem a necessidade de utilizar as classes produtivas do sistema. Através deles, podemos simular comportamentos das classes/métodos originais, mas sem executá-las diretamente.

15.1 Explorando o conceito de dublês de testes

Esse conceito foi descrito por Gerard Meszaros, no livro *XUnit Test Patterns*, como ***Fake***, ***Test Stubs***, ***Spies*** e ***Mocks***. Veja a seguir uma breve descrição de cada um desses termos.

Fake

Fake é um objeto falso que possui a mesma assinatura e os mesmos métodos da classe original, mas não com a mesma implementação. Geralmente os utilizamos para simular o comportamento da classe original.

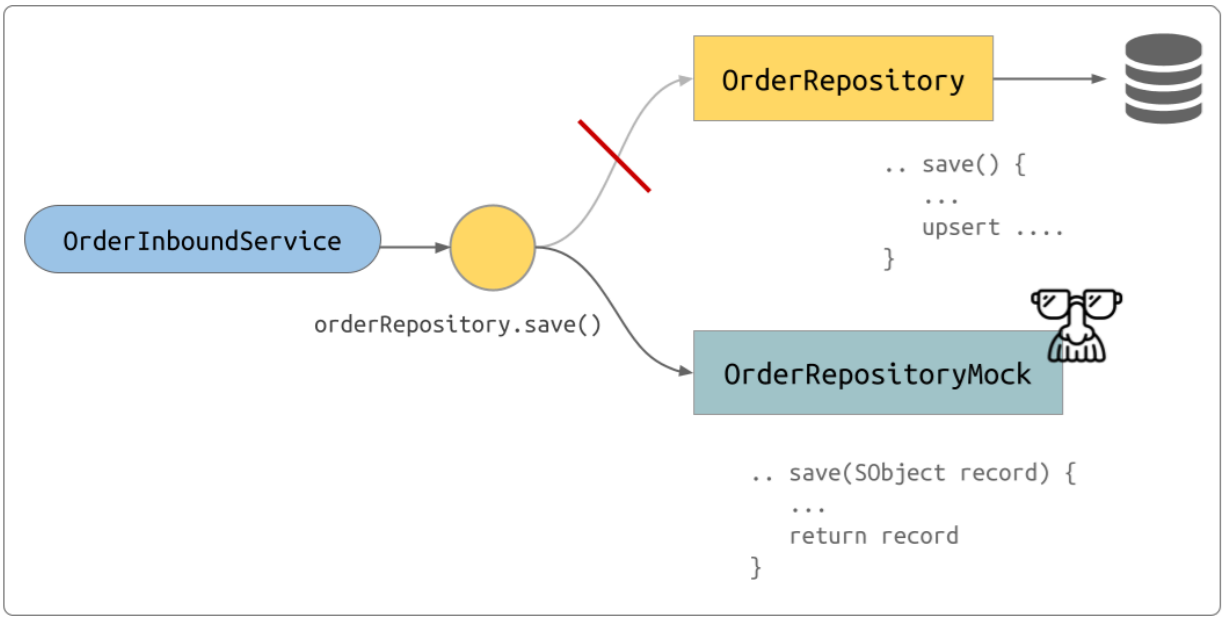


Figura 15.1: Exemplo de Fake

Essa imagem traz um exemplo de utilização de uma classe **Fake**, onde a `OrderRepositoryMock` ("o bigodudo") é um simulador da classe real `OrderRepository`, sobrescrevendo o comportamento do método `save`.

Test Stub

Test Stub (esboço de teste) é um objeto que contém dados predefinidos e os utiliza para atender a chamadas durante os testes. É utilizado quando não podem ser criados ou utilizados os objetos reais, visto que causariam efeitos colaterais.

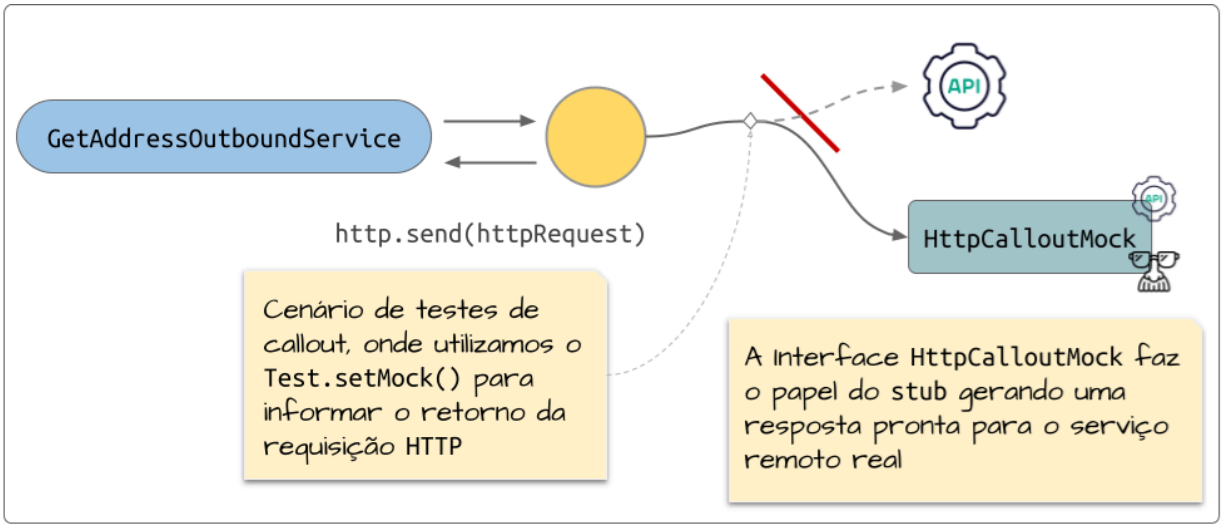


Figura 15.2: Exemplo de Stub

Nós já utilizamos *Stubs* sem saber! Quando testamos serviços externos (*Callout*), utilizamos o `Test.setMock` para injetar um *Stub*, que é a implementação da interface `HttpCalloutMock`. A classe que implementa essa interface retorna um `HttpResponse` pronto para o escopo de teste.

Test Spy

O ***Spy*** é um *Stub* com capacidade de registrar informações sobre a execução.

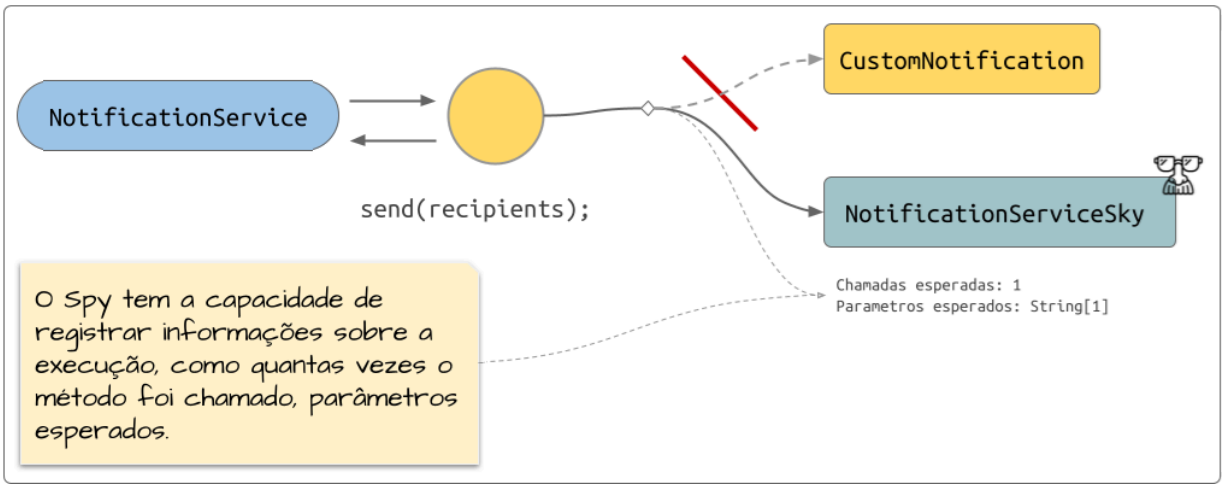


Figura 15.3: Exemplo de Spy

Essa imagem apresenta a principal característica do *Spy*, que é a capacidade de registrar informações da execução, por exemplo, a quantidade de vezes que o método é executado. Esse recurso é interessante para operações que realizam DML, para garantir que não estão ocorrendo dentro de loop.

Test Mock

Podemos dizer que o ***Mock*** é a composição de todos os anteriores, porém com a capacidade de registrar e verificar as chamadas recebidas. Com isso, é possível fazer asserções mais apuradas sobre todo o comportamento. Essa variação de dublê de testes é muito poderosa, pois através dela é possível fazer a simulação de um determinado resultado, avaliar se determinado método foi executado, entre outras funcionalidades.

Recentemente, descobri, no repositório https://github.com/gscloudsolutions/GS-Apex-Mocking-Framework, uma implementação muito interessante e que aplica muito bem esse conceito. Ela utiliza a interface `StubProvider` juntamente com o método de `Test.createStub` para criar os *Mocks*. Essa variação será amplamente explorada mais à frente, mas vou dar um pequeno *spoiler* através do código a seguir:

```
@isTest
public static void shouldSaveCaseAndReturnSuccess () {

    //setup
    Mocker mocker = Mocker.startStubbing();

    CaseRepository mockedRepository;

    // criação do repository "mockado"
    mockedRepository = (CaseRepository)
                               mocker.mock(CaseRepository.class);

    Case caze = CaseFixtureFactory.newCase();
```

```
    Case expected = CaseFixtureFactory.newWithFakeId();

    // setup do mock com o resultado esperado pelo
    // método save
    mocker.when(mockedRepository.save(caze))
                .thenReturn(expected);

    mocker.stopStubbing();

    CaseService service = new CaseService();

    // injeção do mock na classe de serviço
    service.repository = mockedRepository;

    Case storedCase = service.save( caze );

    Assert.areEqual(expected, storedCase);

}
```

O código apresentado nos traz uma nova forma de testar, com muita Orientação a Objetos por "baixo dos panos". De forma resumida, esse código cria um simulador da classe `CaseRepository`, possibilitando o retorno de um `Case` com um `Id` falso e, com isso, é possível testar o serviço sem persistir o registro. Este capítulo abordará algumas etapas cruciais para o entendimento desse código. Então, bora "codar"!

Para facilitar a compreensão vamos convencionar que a palavra *Mock* será utilizada quando temos a intenção de criar nossos dublês de teste. Eles terão inicialmente os comportamentos de ***Stubs*** e ***Fakes***, mas ainda não terão a mesma capacidade de verificação provida pelo framework apresentado.

15.2 Criando nosso primeiro dublê de teste

(Mock)

Para a criação do ***Mock*** de teste, vamos usar e abusar da Orientação a Objetos, pois utilizaremos *herança* e *polimorfismo*. Vamos utilizar o cenário implementado para recebimento de dados de *Pedido* de um sistema externo.

Vamos recordar as dependências da classe `OrderInboundService` utilizando a imagem a seguir:

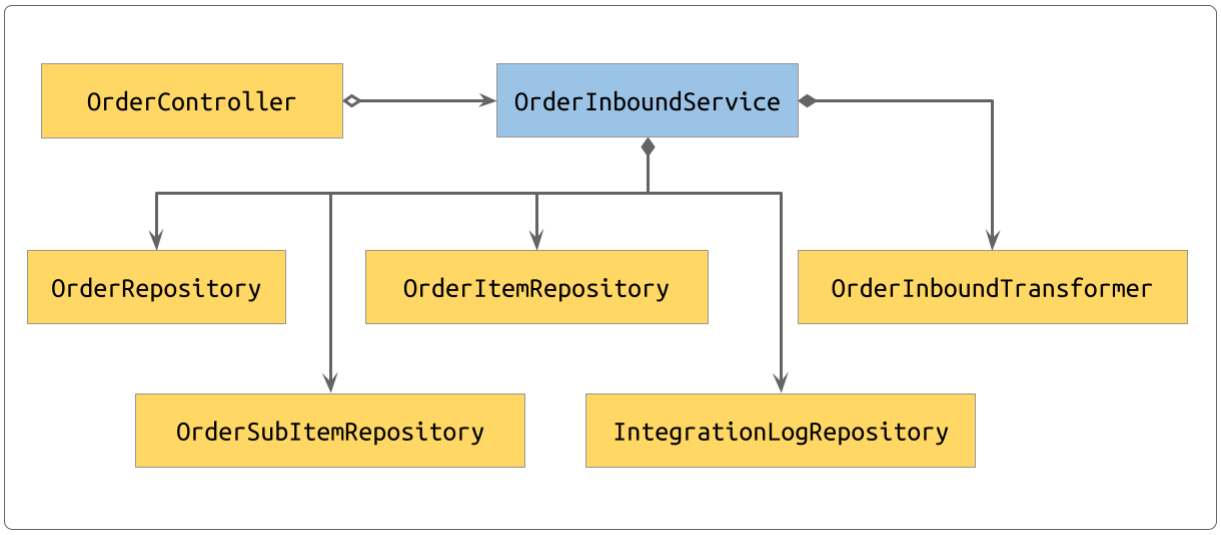


Figura 15.4: Diagrama de classes do OrderInboundService

Com base no diagrama anterior, vamos focar, inicialmente, nas dependências da classe `OrderInboundService`, que são basicamente classes de repositório (**Repository**) e de **Transformer**. Para isso, vamos repassar novamente pelo código do método principal, `save`, da classe `OrderInboundService`.

Ele realiza a transformação do `PurchaseOrderInboundRequest` em `Order`, em `OrderLineItems` e em `OrderSubItems`, e os salva retornando um `PurchaseOrderInboundResponse` com o `Id` da `Order`:

```
public PurchaseOrderInboundResponse save (
                PurchaseOrderInboundRequest purchaseOrder ) {
```

```
    Order order = transformer.toOrder(purchaseOrder);

    List<OrderItem> orderItems =
                    transformer.toOrderItems( purchaseOrder );

    List<OrderSubItem__c> orderSubItems =
                transformer.toOrderSubItems( purchaseOrder );

    this.orderRepository.save(order);
    this.orderItemRepository.save (orderItems);
    this.orderSubItemRepository.save (orderSubItems);

    return new PurchaseOrderInboundResponse( order.Id );

}
```

Com base no código anterior, observe a utilização da classe `OrderRepository`. Quais são suas entradas e saídas? O que é esperado do método `save`, do ponto de vista da classe `OrderInboundService`?

A classe `OrderInboundService` **espera** que uma instância do tipo `OrderRepository`, ao **chamar** o seu método `save`, **retorne** um `Order` com um `Id` válido ou um erro, uma `DmlException`.

#fica-a-dica 🤔

Se a `OrderInboundService` espera que a instância de `OrderRepository` salve o `Order` e retorne um `Id` válido, **então** precisamos de uma classe que simule ser a `OrderRepository`. Esse simulador salvará o conteúdo "de mentirinha" e retornará um `Id` de `Order` válido.

Para fazer isso, precisamos aplicar o "pulo do gato", que é o conceito de "***É um***"! Orientação a Objetos aplicada! Lembre-se: quando utilizamos de herança, a classe filha é a própria classe. Dessa forma,

quando a `OrderRepositoryMock` herda a `OrderRepository`, ela é a própria `OrderRepository`. E é aí que a magia acontece!

Dado que nossa classe `OrderInboundService` espera qualquer implementação que seja do tipo `OrderRepository`, podemos substituí-la por uma classe ***Fake*** que retorne o comportamento desejado.

Nasce o primeiro Mock (Fake)

Code mode on! Vamos criar uma classe chamada `OrderRepositoryMock` que estenderá a classe `OrderRepository` e sobrescreverá (*override*) o método `save` que está implementado em `AbstractRepository`. Nossa primeira implementação será bem simples, só imprimirá no console e retornará a instância do `Order` recebida.

> **#se-liga-na-info** 😎
>
> Só será possível sobrescrever classes e métodos que tenham o modificador `virtual`. Então vamos modificar a classe `OrderRepository` para que ela possa ser herdada.

```
public virtual class OrderRepository extends AbstractRepository {

    // outros métodos foram omitidos

}
```

Em seguida, vamos criar a sua classe `Mock` com um código bem simples. Ela deve ser anotada como `@isTest`, pois só deverá ser consumida em escopo de testes:

```
@isTest
public class OrderRepositoryMock extends OrderRepository {

    override
    public Order save ( Order order ) {
```

```
        System.debug ('Mocked save record');

        return order;
    }
}
```

Caiu uma lágrima aqui! Fizemos o nosso primeiro *Mock*, só precisamos agora ajustar a classe `OrderInboundService` para que tenha capacidade de receber uma nova instância da classe `OrderRepository`. Essa nova instância será passada através do novo método `setOrderRepository`:

```
public class OrderInboundService {

    OrderRepository orderRepository;

    // outras variáveis foram omitidas

    public OrderInboundService () {

        this.orderRepository = new OrderRepository();

        // outras inicializações forma omitidas

    }

    //4 injection
    public void setOrderRepository (
                          OrderRepository orderRepository ) {

        this.orderRepository = orderRepository;

    }
```

Ficou *chique*! Podemos passar uma nova instância de uma classe que **seja um** `OrderRepository`.

#fica-a-dica 🤔

Essa técnica é chamada de **injeção de dependência**, pois uma classe externa tem a capacidade de determinar qual instância de uma dependência será utilizada.

Vamos para a classe de teste `OrderInboundServiceTest` ! Ela será inicialmente implementada da forma mais simples, para depois irmos ajustando até que ela esteja completa.

Primeiramente, em nosso método de teste de sucesso, vamos criar uma instância da classe `OrderInboundService` e depois chamar o método `setOrderRepository` , que deverá receber a nova instância da classe `OrderRepositoryMock` que contém o método `save` sobrescrito:

```
@isTest
static void givenPurchaseOrderInboundWhenSaveThenCreateOrder() {

    // given
    PurchaseOrderInboundRequest request;


    request = PurchaseOrderInboundFixtureFactory
                            .newPurchaseOrderInboundRequest();

    OrderInboundService service = new OrderInboundService();

    //aqui que a magia acontece:
    //redeterminamos a instância utilizada
    service.setOrderRepository( new OrderRepositoryMock() );

    // when
    Test.startTest();

    PurchaseOrderInboundResponse response = service
                                                .save(request);
```

```
        Test.stopTest();

        // then
        Assert.isNotNull( response.id
                , 'Esperado um response com um Id válido' );

}
```

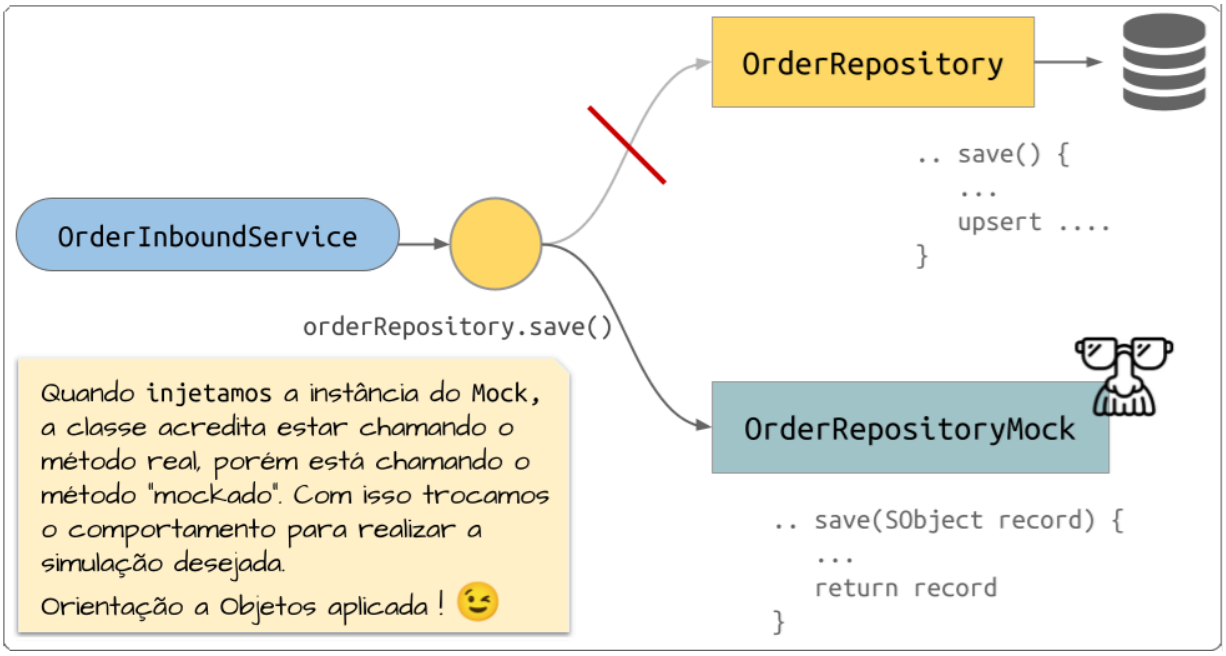


Figura 15.8: Fazendo mock da classe OrderRepository

O código anterior mostrou como foi simples de trocar a instância de `OrderRepository` pela sua versão "mockada", mas, se rodarmos esse teste, ele falhará! Isso porque estamos esperando um `response.id` diferente de nulo. Esse erro ocorre devido ao nosso teste "mockado" não retornar um `Order.Id` válido. Mas como retornar um `Id` válido em um teste "mockado"?

Em cenários de testes "mockados", e somente neles, podemos utilizar um artifício, que é fazer uma pesquisa na Org do objeto em questão (que, no caso, é o `Order`) e, com isso, montar uma String JSON com esse `Id` . Isso pode ser obtido rodando um script de forma anônima, como o a seguir:

```
System.debug(
    JSON.serialize( [SELECT Id FROM Order LIMIT 1]));
```

Ele retornará no console um JSON semelhante a este: `{"Id": "8018c000001hsePAAQ"}`. Com esse truque, podemos ajustar o nosso *Mock* para receber esse JSON e retornar uma instância válida de `Order` com um `Id` também válido:

```
@isTest
public class OrderRepositoryMock extends OrderRepository {

    String payload;

    //recebe uma String JSON para ser utilizada
    //nos métodos sobrescritos
    public OrderRepositoryMock (String payload) {
        this.payload = payload;
    }

    override
    public Order save ( Order record ) {

        System.debug ('Mocked save record');

        //JUMP OF THE CAT ;)
        Order order = (Order)
                        JSON.deserialize ( this.payload
                                            , Order.class );
        record.Id = order.Id;

        return record;
    }

}
```

#fica-a-dica 🤔

Observe que estamos passando uma variável `String payload` como parâmetro do **construtor** da classe `OrderRepositoryMock`. Esse é **um superpulo do gato** para passar como parâmetro o conteúdo desejado. Ele poderá ser retornado pelos métodos "mockados" da classe.

Estamos quase lá! Só falta ajustar a classe de teste para utilizar a nova implementação de `OrderRepositoryMock`:

```
@isTest
static void givenPurchaseOrderInboundWhenSaveThenCreateOrder() {

    // given
    PurchaseOrderInboundRequest request;

    request = PurchaseOrderInboundFixtureFactory
                        .newPurchaseOrderInboundRequest();

    String savedOrderPayload
                = '{"Id": "8018c000001hsePAAQ"}';

    OrderInboundService service
                            = new OrderInboundService();

    //pulo do gato ;)
    service.setOrderRepository(
            new OrderRepositoryMock( savedOrderPayload ));

    // when
    Test.startTest();

    PurchaseOrderInboundResponse response
                                    = service.save( request );
```

```
        Test.stopTest();

        // then
        System.assert( response.id != null
                , 'Esperado um response com um Id válido' );

    }
```

Passou liso! Testes rodando! Mas, apesar de tudo estar rodando e de forma extremamente rápida, nossa implementação está difícil de consumir, por nos obrigar a trabalhar com uma String JSON só para ter um `Id` válido. A utilização de uma String JSON funciona muito bem para ser passada em nosso *Mock* como o resultado de uma consulta, mas para representar o valor persistido não. É aí que entra a classe utilitária `IdGenerator` !

Melhorando nossos Mocks com IdGenerator

A `IdGenerator` preencherá nossa lacuna nos testes "mockados", criando um `Id` ***fake*** válido.

Apenas para você ter uma ideia de como a `IdGenerator` foi implementada, disponibilizo aqui parte de sua implementação, mas não se preocupe em analisar o seu código por completo, dada a sua complexidade. Você pode obtê-lo na íntegra no repositório, em: https://github.com/ercarval/salesforce-book-snippets/.

O `Id` do Salesforce é unívoco em todas as Orgs. Ele é composto por 18 caracteres. O código a seguir trata essa geração de forma bem resumida, considerando alguns pontos como:

- Determinação do prefixo do `SObject` , que são os três primeiros caracteres do Objeto Salesforce;
- O `Id` da Org, que obtemos pelo `Id` do usuário corrente;
- Geração randômica dos dígitos restantes.

```
/**
 * Based on https://salesforce.stackexchange.com/questions/1653/what-are-
```

```
salesforce-ids-composed-of
 * Credits for https://www.gammone.com/programmazione/come-convertire-id-
salesforce-da-15-a-18-caratteri
 *
 * @author eduardo.bisso
 */
@isTest
public class IdGenerator {

    static final Integer PREFIX_LENGTH = 3;
    static final Integer ORG_INSTANCE_RESERVED_LENGTH = 5;

    static final String BASE62_ALPHABET =
'0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ';

    static Integer sequence = 0;

    public static SObject generate ( SObject record ) {

        record.Id = IdGenerator
                    .generate( record.getSObjectType() );

        return record;
    }

    public static String generate (String objectPrefix) {

        String prefix = objectPrefix
                            + getOrgPrefix()
                            + '00000'; //

        String id = prefix + buildRandom() + (++sequence);

        return appendSufix(id);

    }

    static String getOrgPrefix () {
        return UserInfo.getUserId()
```

```
            .substring(PREFIX_LENGHT
                   , ORG_INSTANCE_RESERVED_LENGTH);
}

static String buildRandom() {

    String sequence = '';

    for ( Integer i = 0; i < 4; i++) {

        Integer idx = Math.mod(
                        Math.abs (Crypto.getRandomInteger())
                        , BASE62_ALPHABET.length());

        sequence += BASE62_ALPHABET.substring ( idx, idx+1);
    }

    return sequence;

}

static String appendSuffix (String id) {

    String suffix = '';
    String baseChars = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ012345';

    for(Integer i = 0; i < 3; i++) {

        Integer flags = 0;

        for(Integer j = 0; j < 5; j++) {

            String c = id.substring( i * 5 + j
                                   , i * 5 + j + 1);

            if ((c.compareTo('A')>=0)
                && (c.compareTo('Z')<=0)) {
                flags += 1 << j;
            }
```

```
            }

            suffix += baseChars.substring(flags, flags+1);
        }

        return id + suffix;

    }

}
```

Bastante código, como comentei, mas não se preocupe com tentar entendê-lo, você pode simplesmente consumi-lo. Mas recomendo que volte nele futuramente para um entendimento maior de como são criados os `Id(s)` dentro da plataforma.

Vamos ajustar a nossa classe `OrderRepositoryMock` para utilizar a `IdGenerator`:

```
@isTest
public class OrderRepositoryMock extends OrderRepository {

    String payload;

    public OrderRepositoryMock () {
    }

    public OrderRepositoryMock (String payload) {
        this();
        this.payload = payload;
    }

    override
    public  SObject save ( SObject record
                        , Schema.SObjectField field ) {

        //gera o id e o mapeia para a variável record
```

```
        return IdGenerator.generate(record);
    }

}
```

#agora-o-jogo-vira

Como ficou elegante, limpa e mais fácil de consumir essa nova implementação de ***Mock***. Para finalizar a nossa classe de testes, só precisamos ajustá-la para utilizar o construtor `OrderRepositoryMock()`, sem argumentos:

```
@isTest
static void givenPurchaseOrderInboundWhenSaveThenCreateOrder() {

    // given
    PurchaseOrderInboundRequest request;


    request = PurchaseOrderInboundFixtureFactory
                        .newPurchaseOrderInboundRequest();

    OrderInboundService service
                            = new OrderInboundService();

    //jump of the cat ;)
    service.setOrderRepository(
            new OrderRepositoryMock());

    // when
    Test.startTest();

    PurchaseOrderInboundResponse response
                                    = service.save( request );

    Test.stopTest();

    // then
```

```
    System.assert( response.id != null
            , 'Esperado um response com um Id válido' );

}
```

Observe na imagem a seguir o comportamento do *Mock* em tempo de execução, ao substituir a implementação da classe original pela "mockada":

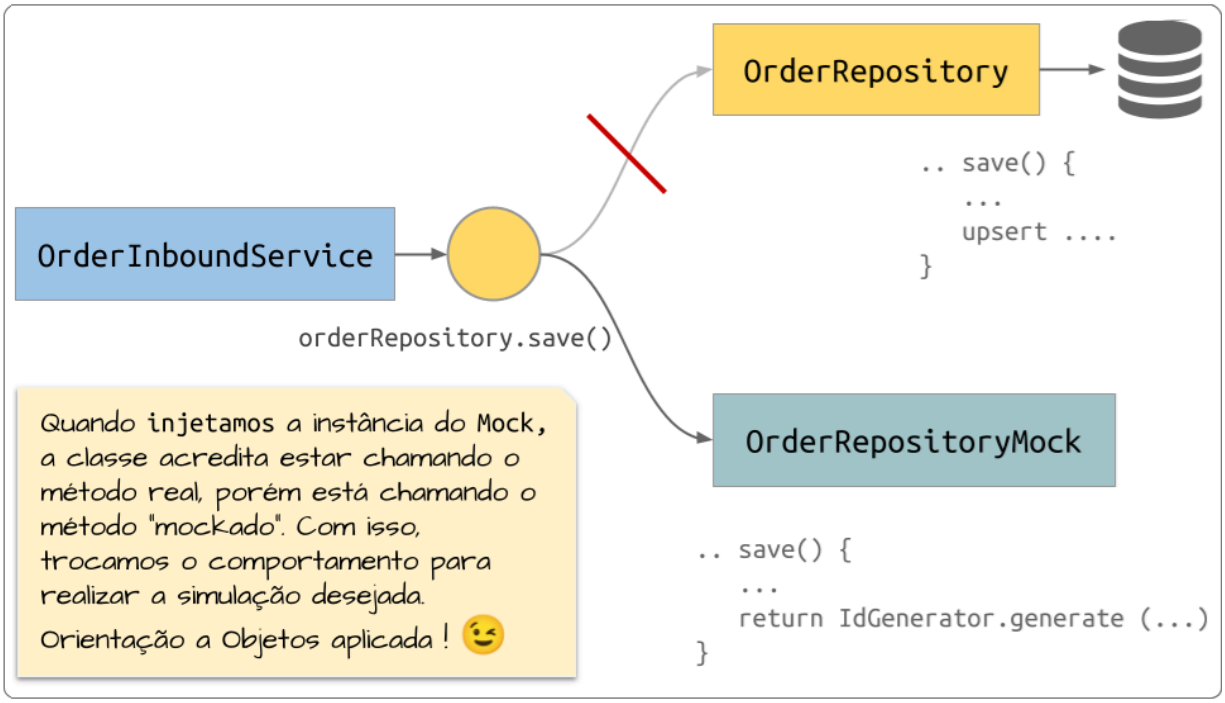


Figura 15.11: Mock com generator

Finalizando nosso teste 100% "mockado"

Agora é replicar a mesma ideia para as outras classes de `Repository`. Para isso, precisamos garantir que as classes que serão "mockadas", bem como seus métodos, estejam como `virtual`, para que possam ser sobrescritos.

Observe novamente a implementação de `AbstractRepository` e `OrderLineItemRepository`, onde adicionaremos o modificador `virtual` para que os métodos possam ser sobrescritos:

```
/**
 * @author eduardo.bisso - dev-builder
 */
public virtual class AbstractRepository {

    // outros métodos omitidos
    virtual
    public List<SObject> save (List<SObject> records
                               , Schema.SObjectField field ) {

        if ( records == null || records.isEmpty() ) {
            return records;
        }

        Database.upsert(records, field);

        return records;

    }

}

public virtual class OrderLineItemRepository
                                    extends AbstractRepository {

    Schema.SObjectField ORDER_ITEM_EXTERNAL_ID =
                                Schema.OrderLineItem.ErpCode__c
                                            .getDescribe()
                                            .getSObjectField()

    // outros métodos omitidos
    // adicionado o modificador virtual
    virtual
    public List<OrderItem> save ( List<OrderItem> orderItems ) {

        save (orderItems, ORDER_ITEM_EXTERNAL_ID);

        return orderItems;
    }
```

```
}
```

Com todas as classes prontas para extensão, podemos "mockar" o método `save` da `AbstractRepository`, dado que é ele que está realizando a operação DML de fato:

```
@isTest
public class OrderItemRepositoryMock
                            extends OrderItemRepository {

    String payload;

    public OrderLineItemRepositoryMock () {
    }

    public OrderLineItemRepositoryMock (String payload) {
        this();
        this.payload = payload;
    }

    override
    public List<OrderItem> save ( List<OrderItem> orderItems ) {

        IdGenerator.generate(orderItems);

        return orderItems;
    }

}

@isTest
public class OrderSubItemRepositoryMock
                            extends OrderSubItemRepository {

    String payload;

    public OrderSubItemRepositoryMock () {
    }
```

```
    public OrderSubItemRepositoryMock (String payload) {
        this();
        this.payload = payload;
    }

    override
    public List<OrderSubItem__c> save ( List<OrderSubItem__c>
orderSubItems ) {

        IdGenerator.generate(orderSubItems);

        return orderSubItems;
    }

}
```

Mocks prontos! Vamos fechar o nosso teste:

```
@isTest
static void givenPurchaseOrderInboundWhenSaveThenCreateOrder() {

    // given
    PurchaseOrderInboundRequest request;


    request = PurchaseOrderInboundFixtureFactory
                        .newPurchaseOrderInboundRequest();

    OrderInboundService service
                            = new OrderInboundService();

    //jump of the cat ;)
    service.setOrderRepository(
            new OrderRepositoryMock());

    service.setOrderItemRepository(
            new OrderItemRepositoryMock());

    service.setOrderSubItemRepository(
```

```
            new OrderSubItemRepositoryMock());

        // when
        Test.startTest();

        PurchaseOrderInboundResponse response
                                        = service.save( request );

        Test.stopTest();

        // then
        System.assert( response.id != null
                , 'Esperado um response com um Id válido' );

}
```

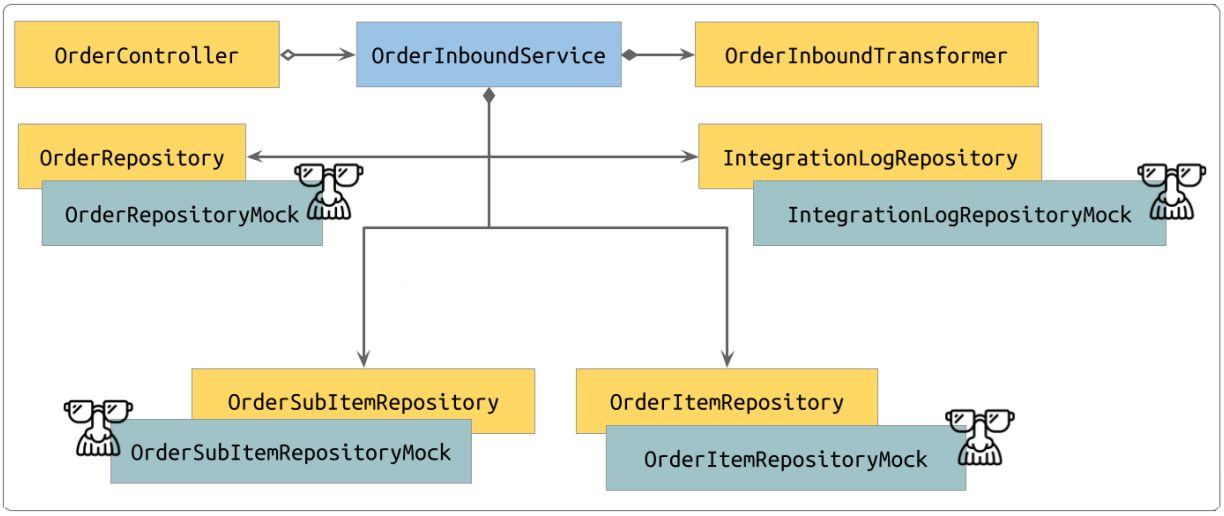


Figura 15.12: OrderInboundService com seus mocks

Através do código anterior temos uma implementação de *Mock* que realiza todo o fluxo de criação de pedidos, porém sem persistir nada no banco. Com isso, o teste roda muito mais rápido.

Atendemos plenamente ao princípio ***Fast*** de testes de unidade. Inicialmente temos mais código para ser implementado, mas as próximas classes de teste que consumirão esses *Mocks* serão muito mais simples de serem feitas.

#fica-a-dica 🤔

Utilizo as classes de teste de `Repository` para deixar uma versão padrão de *Mock*; dessa forma, todas as classes dependentes desse `Repository`, quando forem fazer um teste "mockado", poderão reutilizar essa implementação.

Testando fluxos de erro com Mock

Não basta só testar o cenário de sucesso, também temos que simular todos os cenários de falha. Para isso, basta passar um novo *Mock* que forneça uma implementação que lance uma `Exception`, como `DmlException` ou qualquer outra.

No próximo exemplo, vamos criar um *Mock* específico para erros durante o `save`. Para isso, vamos forçar uma exceção do tipo `DmlException`:

```
@isTest
public class OrderRepositoryDmlExceptionMock
                                    extends OrderRepository {

    public OrderRepositoryDmlExceptionMock () {
    }

    override
    public  SObject save ( SObject record
                        , Schema.SObjectField field ) {

        if (true) { // para enganar o compilador
            throw new DmlException ('Invalid Account');
        }

        return null;
    }
```

```
}

@isTest
static void
givenPurchaseOrderWithInvalidAccountWhenSaveThenThrowDmlException () {

    // given
    PurchaseOrderInboundRequest request;

    request = PurchaseOrderInboundFixtureFactory
                        .newPurchaseOrderInboundRequest();

    OrderInboundService service
                            = new OrderInboundService();

    //jump of the cat ;)
    service.setOrderRepository(
            new OrderRepositoryDmlExceptionMock());

    service.setOrderItemRepository(
            new OrderItemRepositoryMock());


    service.setOrderSubItemRepository(
            new OrderSubItemRepositoryMock());

    service.setPurchaseOrderInboundTransformer(
                    new PurchaseOrderInboundTransformerMock());
    // when
    Test.startTest();

    PurchaseOrderInboundResponse response;

    try {

        response = service.save( request );

        Assert.fail ('Deve lançar uma DmlException');
```

```
    } catch (DmlException e) {

        Assert.isTrue (true);

    }

    Test.stopTest();

}
```

Através das classes desse código, você pode observar que podemos embutir o comportamento desejado em nossas classes de *Mock* e, assim, criar os nossos cenários de testes de sucesso e de falha.

Observações finais

#o-que-não-posso-fazer-se-não-perco-os-dedos

Quando trabalhamos com testes "mockados", **não podemos, em hipótese alguma, persistir o dado**, ou seja, **operações REAIS de DML não podem acontecer**. Então você **SEMPRE DEVE "MOCKAR"** os métodos `save` de classes `Repository`. Caso você se esqueça de fazer isso, terá problemas na implantação de sua solução em outro ambiente.

Todo esse processo de *Mock* só é possível porque estamos utilizando **polimorfismo**. Graças a essa característica da Orientação a Objetos, podemos trocar o comportamento real por um **fake**, possibilitando a substituição do comportamento. Dessa forma, devemos abolir o uso de métodos com o escopo *estático*, pois eles não proporcionam esse comportamento.

Lembre-se: uma classe que está difícil de testar indica que há problemas de design nessa classe.

Resumão de pontos dos quais você não pode se esquecer de fazer em classes produtivas para que elas possam ser testadas de forma "mockada":

- Classe e métodos devem ser `virtual`, possibilitando a herança e sobrescrita;
- Utilizar métodos *setters* para possibilitar a injeção de dependência no escopo de testes;
- As dependências de classes produtivas devem ser criadas como atributos/variáveis de instância;
- Não utilizar métodos estáticos, pois não possibilitam a reescrita.

Agora que vimos como criar testes 100% "mockados", podemos dar mais um passo, onde veremos como podemos utilizar essa prática para outros tipos de classe e como podemos compor testes com o framework apresentado, juntamente com testes *fakes*.

Capítulo 16
Utilizando o framework Mocker

Chegou a hora de adicionar mais um *Jutsu* ao arsenal de práticas de teste. Você já ganhou um *spoiler* durante o capítulo 15, *Mocks*, onde vimos um pequeno exemplo do framework *Mocker*. Com esse novo *Jutsu*, teremos maior capacidade de criar testes "mockados" com menos esforço. Preparei uma pequena lista de benefícios em sua utilização:

- Não temos a necessidade de criar novas classes para realização do *Mock*;
- Capacidade de criar *Test Spy*, que permite avaliar a quantidade de vezes que um método foi executado;
- Capacidade de criar *Test Stub*, semelhante ao que vimos no capítulo anterior, porém sem a necessidade de criar novas classes;
- Capacidade de criar cenários de falha de forma simples.

Com todas essas vantagens em mente, vamos "virar esse framework do avesso" e ver todos os seus recursos.

16.1 Introdução ao framework

Os frameworks de *Mock*, independente da linguagem, seguem um padrão de execução que é conhecido pelo acrônimo em inglês ***AAA*** (*Arrange, Act and Assert*), que, em português, seria *organizar*, *agir* e *avaliar/afirmar*.

Se observarmos nossos testes anteriores, nós fazemos exatamente isso, porém utilizando a notação de *given/when/then* do BDD, onde

temos uma etapa de configuração do *Mock* ou *setup* de dados, seguida da execução da classe que será testada, e finalizamos com nossas asserções.

O que muda em todos os modos de testar é a etapa de configuração. Para testes integrados, nós criamos dados no banco para a execução da classe que será testada. Já para testes "mockados", sem a utilização do framework, nós parametrizamos nossas classes de *Mock* com os dados *fake* para retornar um conteúdo esperado. Com o framework `Mocker`, faremos uma etapa muito semelhante aos nossos testes "mockados" anteriores, porém com menos classes.

Delimitando o escopo do que será "mockado"

A etapa de registro/configuração é uma etapa fundamental para a boa execução de nosso teste "mockado". Nela determinamos o comportamento simulado desejado. Com o framework, essa etapa é delimitada pelos comandos `Mocker mocker = Mocker.startStubbing()` e `mocker.stopStubbing()`. Observe o código a seguir onde iniciaremos o ciclo de gravação do *Mock*:

```
Mocker mocker = Mocker.startStubbing();

ContractRepository mockedRepository = (ContractRepository)
                        mocker.mock(ContractRepository.class);

mocker.stopStubbing();
```

O método `Mocker.startStubbing()` é o responsável por criar a instância do `mocker` propriamente dito. É através dessa instância que registramos todos as classes "mockadas" e o comportamento desejado. Já o método `mocker.stopStubbing()` finaliza a etapa de gravação do comportamento simulado.

Quando utilizamos o método `mocker.mock()`, estamos registrando a classe que desejamos "mockar" e consequentemente podemos iniciar a parametrização da classe. As classes que serão "mockadas"

também necessitam de configuração do resultado desejado durante a execução do teste, semelhante ao que fizemos no capítulo anterior.

16.2 Arrange: parametrizando a execução

Durante a etapa de gravação do comportamento desejado, utilizamos os métodos `when` e `then`, expostos pelo `Mocker`. Veja um exemplo simples onde simulamos a execução do método `findByIds` de `ContractRepository`.

```
// ----- Setup
Mocker mocker = Mocker.startStubbing();

ContractRepository mockedRepository = (ContractRepository)
                        mocker.mock(ContractRepository.class);

// Dado
String contractId = IdGenerator.generate (
                                Contract.getSObjectType() );

List<String> originalContractIds = new List<String> {contractId};

Contract expectedOriginalContract = ContractFixtureFactory
                                .newSignedContract(contractId);

List<Contract> expectedContract = new List<Contract> {
                                    expectedOriginalContract };

//Quando o método findByIds for executado
//Então ele deve retornar um determinado resultado
mocker.when ( mockedRepository.findByIds (originalContractIds) )
        .thenReturn( expectedContract);

mocker.stopStubbing();
```

```
// ----- Execute
List<Contract> storedContracts;

storedContracts = mockedRepository.findByIds (
                                        originalContractIds );

// ----- Assert
Assert.areEquals ( 1 , storedContracts.size() );
Assert.areEquals ( contractId , storedContracts.get(0).Id );
```

Quando utilizamos `mocker.when` , estamos informando ao *Mocker* que, **quando** um método com um parâmetro específico for executado, **então** deverá retornar um determinado resultado, através do `thenReturn` .

Nessa etapa de gravação, podemos gravar diversos *Mocks* e o comportamento desejado para cada um deles, que serão consumidos durante a fase de execução. Após a fase de gravação, que finaliza com o comando `mocker.stopStubbing();` , executamos a operação propriamente dita, que, em nosso exemplo, foi a chamada do método `findByIds (originalContractIds)` .

A forma como configuramos nosso *Mock* só vai retornar a lista de contratos se receber o *input* correto. Para ficar mais claro, observe a implementação a seguir, onde executamos o método `findByIds` com dados não configurados:

```
// ----- Setup
// Dado
List<String> givenContractIds = new List<String> { '8001U00000GiMz2QAF' };

List<Contract> expectedContract = new List<Contract> {
                ContractFixtureFactory.newSignedContract(
                                        givenContractIds.get(0))};

mocker.when ( mockedRepository.findByIds (givenContractIds) )
        .thenReturn( expectedContract);
```

```
mocker.stopStubbing();

// ----- Execute
List<Contract> storedContracts = mockedRepository.findByIds (
                                                 givenContractIds );

// ----- Assert Mocked Content
Assert.areEquals ( 1 , storedContracts.size() );
Assert.areEquals ( contractId , storedContracts.get(0).Id );

// ----- Assert NOT Mocked Content
List<String> notMockedContracts = new List<String> {
                                          '8001U00000Zuoc3QAB' };

List<Contract> otherContracts = mockedRepository.findByIds (
                                                notMockedContracts );

Assert.isTrue ( otherContracts.isEmpty() );
```

Esse código só retornará a lista de contratos se receber a `List<String>` com o `Id = 8001U00000GiMz2QAF`. Esse recurso é interessante porque podemos fazer com que o *Mock* execute o mesmo método com várias alternativas de entrada e, consequentemente, com retornos variados.

Mas, em determinados cenários, podemos configurar o nosso *Mock* para que ignore os parâmetros e sempre retorne o mesmo conteúdo para a execução desse método.

```
List<String> anyArgs = new List<String> {''} ;

List<Contract> expectedContract = new List<Contract> {
            ContractFixtureFactory.newSignedContract(
                                          '8001U00000Zuoc3QAB') } ;

// configuração padrão
mocker.when ( mockedRepository.findByIds ( anyArgs ) )
                            .withAnyValues()
```

```
                            .thenReturn( expectedContract );

mocker.stopStubbing();
```

Essa estratégia é válida para cenários nos quais queremos avaliar somente o resultado da execução, não importando o parâmetro passado. Ela também pode ser combinada: caso não encontre o parâmetro passado, o *Mock* sempre retornará a configuração padrão. Veja o exemplo a seguir:

```
List<String> givenContractIds = new List<String> {
                                        '8001U00000GiMz2QAF' };

List<Contract> specificContracts = new List<Contract> {
                ContractFixtureFactory.newSignedContract(
                                       givenContractIds.get(0))};

// configuração específica
mocker.when ( mockedRepository.findByIds (givenContractIds) )
        .thenReturn( specificContracts );


List<Contract> otherContracts = new List<Contract> {
                ContractFixtureFactory.newSignedContract(
                                        '8001U00000Zuoc3QAB' )};

List<String> anyArgs = new List<String> {''} ;

// configuração padrão
mocker.when ( mockedRepository.findByIds ( anyArgs ) )
                            .withAnyValues()
                            .thenReturn( otherContracts );

mocker.stopStubbing();

// retorna o conteúdo para o Id específico
List<Contract> storedContracts = mockedRepository.findByIds (
                                                 givenContractIds );
```

```
Assert.areEquals ( specificContracts , storedContracts );

List<String> anyIds = new List<String> {
                                          '8001U00000Zuoc3QAB' };

// o mesmo retorno para qualquer Id enviado.
List<Contract> otherStoredContracts =
                        mockedRepository.findByIds ( anyIds );

Assert.areEqual ( 1,  otherContracts.size() );
Assert.areEqual ( otherContracts ,  defaultContracts );
```

Através de configurações como essa, ganhamos uma alta capacidade de estruturar de forma simples nossos cenários de teste.

16.3 Act: executando o Mock

Essa etapa já foi apresentada anteriormente, mas sem o destaque necessário. É nela onde executamos o *Mock* e obtemos o resultado de nossos métodos. No caso do *Mocker*, executamos logo após a finalização gravação utilizando o comando `mocker.stopStubbing()`. O código a seguir apresenta a execução do método "mockado" `mockedRepository.findByIds`, que retorna a lista de contratos válida:

```
// Arrange
Mocker mocker = Mocker.startStubbing();

// setup do mock omitida
mocker.stopStubbing();

// Act
List<Contract> storedContracts = mockedRepository.findByIds (
                                                 givenContractIds );
```

Dependendo da característica de nossos testes, podemos ter a chamada de mais de uma operação para a realização de nossos

testes. Essa é mais uma das vantagens do *Mocker*: através dele, podemos configurar os nossos *Mocks* de forma mais granular. Com isso, podemos executar a mesma chamada de método obtendo um comportamento diferente, como no código apresentado a seguir, onde executamos o método `findByIds` mais de uma vez:

```
// ---- Arrange
// Dado
List<String> givenContractIds = new List<String> { '8001U00000GiMz2QAF' };

List<Contract> expectedContract = new List<Contract> {
                ContractFixtureFactory.newSignedContract(
                                        givenContractIds.get(0))};

mocker.when ( mockedRepository.findByIds (givenContractIds) )
        .thenReturn( expectedContract);

mocker.stopStubbing();

// ----- Act
List<Contract> storedContracts = mockedRepository.findByIds (
                                            givenContractIds );

// ----- Assert
Assert.areEquals ( 1 , storedContracts.size() );
Assert.areEquals ( contractId , storedContracts.get(0).Id );

// ----- Assert NOT Mocked Content
List<String> notMockedContracts = new List<String> {
                                        '8001U00000Zuoc3QAB' };

List<Contract> otherContracts = mockedRepository.findByIds (
                                            notMockedContracts );

Assert.isTrue ( otherContracts.isEmpty() );
```

Esse código é bem simples, mas demonstra bem a característica do framework de avaliação e execução de operações "mockadas", que é uma vantagem sobre a utilização de classes de teste com ***Fake***

Mock. Isso porque passamos a informação que deverá ser retornada no construtor, sem a avaliação do dado informado.

16.4 Assert: validando a execução

Nos exemplos anteriores, nós fizemos asserções somente do conteúdo retornado, isto é, se o método retornou a lista com o `Id` correto, mas não quantas vezes uma determinada operação foi executada. Esse tipo de capacidade pode ser interessante para saber se determinado algoritmo pode estourar os limites da plataforma.

Veja o exemplo a seguir da classe `DraftContractEnricher`, onde temos um possível problema de limite:

```
public class DraftContractEnricher {

    public List<Contract> createDraftContracts (
                                    List<Account> accounts ) {

        Map<String, List<Opportunity>> openOpportunities =
                            findOpenOpportunities (accounts);

        List<Contract> allContracts = new List<Contract>();
        for (String accountId : openOpportunities.keySet() ) {

            List<Contract> contracts = buildContracts (
                        openOpportunities.get(accountId) );

            allContracts.addAll ( save (contracts) );
        }

        return allContracts;

    }

    Map<String, List<Opportunity>> findOpenOpportunities (
```

```
                                        List<Account> accounts ) {
        // pesquisa e conversão para Map serão omitidos
    }

    List<Contract> save ( List<Contract> contracts ) {
        return (List<Contract>) repository.save (contracts);
    }
}
```

O exemplo apresentado é bem simples e traz um erro comum para iniciantes na plataforma, que é a realização de operações DML/SOQL dentro de um *loop*. Vamos utilizar o *Mocker* para prevenir que erros de implementação como esse ocorram.

Em nosso método de teste, vamos garantir que tanto os métodos de `findOpenOpportunities` quanto o de `save` ocorram apenas uma única vez. Para isso, vamos "mockar" a própria classe `DraftContractEnricher` simulando a execução desses dois métodos. Veja a implementação a seguir:

```
Mocker mocker = Mocker.startStubbing();

OpportunityRepository opportunityRepositoryMock;

opportunityRepositoryMock = (OpportunityRepository)
                        mocker.mock(OpportunityRepository.class);

Integer numberOfAccounts = 9;

List<Account> accounts =
  AccountFixtureFactory.newAccountsWithFakeId (numberOfAccounts);

Integer numberOfOpportunities = 20;

List<Opportunity> openOpportunities;

openOpportunities =
    OpportunityFixtureFactory.newOpenOppotunities ( accounts
                                    , numberOfOpportunities );
```

```
mocker.when ( opportunityRepository.findOpenOpportunities (
                                    Lists.toSetIds(accounts) ) )
                .thenReturn( openOpportunities)
                .shouldBeCalledOnce();

List<Contract> expectedContracts =
    ContractFixtureFactory.newDraftContracts (openOpportunities);

ContractRepository contractRepositoryMock;

contractRepositoryMock = (ContractRepository)
                          mocker.mock (ContractRepository.class);

mocker.when ( contractRepository.save ( new List<Contract> ()) )
                    .withAnyValues()
                    .thenReturn( expectedContracts )
                    .shouldBeCalledOnce();

mocker.stopStubbing();

DraftContractEnricher enricher = new DraftContractEnricher();
enricher.repository = contractRepository;
enricher.opportunityRepository = opportunityRepository;

List<Contract> storedContracts;

storedContracts = enricher.createDraftContracts (accounts);

mocker.assert(); // -- aqui está o pulo do gato

Assert.isFalse ( storedContracts.isEmpty() );
```

Em nosso exemplo, utilizamos o método `shouldBeCalledOnce` para indicar que os métodos deveriam ser chamados uma única vez. Os métodos `shouldBeXXX` informam ao *Mocker* que ele também deve registrar quantas vezes determinado método será executado.

Para realizar a asserção dessas validações, devemos utilizar `mocker.assert()`, que avaliará todos os métodos "mockados" e suas restrições lançando uma `AssertException`, que indica o comportamento não esperado. Podemos utilizar outros métodos, como:

- `shouldBeCalled(Integer times)`: parametriza quantas vezes determinado método será executado;
- `shouldNeverBeCalled`: parametriza que o método nunca será executado. Essa opção é interessante em testes de algoritmos de fluxos alternativos, onde o método só será executado se passar em uma determinada condição;
- `shouldBeCalledBetween(Integer minTimes, Integer maxTimes)`: essa opção de configuração nos traz uma capacidade de determinar um intervalo de execução.

O que todas essas opções nos trazem? Flexibilidade e versatilidade para nossos cenários de teste, como o que foi apresentado anteriormente, em que os métodos que realizam operações com o banco sejam executados uma única vez.

A propósito, vou deixar a correção da classe com erro de limite para você :). Fica aqui o desafio.

16.5 Trabalhando com erros

Durante o capítulo anterior, para criar um fluxo de erro era necessário criar uma outra classe, como a `OrderRepositoryDmlExceptionMock`, para que pudéssemos simular os cenários de falha. Com o *Mocker*, podemos fazer isso de forma muito mais simples: basta adicionar o método `thenThrow` em nossa etapa de configuração. Veja o exemplo onde simulamos um erro durante a etapa de gravação dos *Contratos*:

```
Mocker mocker = Mocker.startStubbing();

ContractRepository contractRepositoryMock;

contractRepositoryMock = (ContractRepository)
                          mocker.mock (ContractRepository.class);

// outros mocks omitidos
mocker.when ( contractRepositoryMock.save (new List<Contract>()))
             .thenThrow (
                    new DmlException ('UNABLE TO LOCK ROW')
            );

mocker.stopStubbing();

try {
    // vai gerar um erro
    enricher.createDraftContracts (accounts);

    Assert.isFail();

} catch ( DmlException e ) {

    Assert.isTrue ( e != null );

}
```

Através de alguns exemplos, você pôde ver como podemos reduzir o volume de classes de teste que criamos e como podemos produzir cenários de teste de forma mais rápida, sempre alinhado aos princípios do F.I.R.S.T.

Para fechar com chave de ouro a nossa jornada de testes, veremos no próximo capítulo como podemos "mockar" diversas classes que construímos até aqui.

CAPÍTULO 17
Mockando tudo

Chegou a hora de aplicar todo o nosso arsenal de *Jutsus* de teste. Passaremos pelos principais padrões abordados e como testá-los de forma "mockada".

17.1 Mockando classes Enricher

No capítulo 6, *Definindo papéis e responsabilidades*, quando menciono o padrão **`Enricher`**, implementamos a classe `InactivateOriginalContractEnricher`, que tem a responsabilidade de inativar o *Contrato* anterior pelo novo recém-assinado. Para refrescar a sua memória, veja novamente a classe `InactivateOriginalContractEnricher`:

```
public class InactivateOriginalContractEnricher {

    ContractRepository repository;

    ContractFilter filter;

    public InactivateOriginalContractEnricher () {
        this.repository = new ContractRepository ();
        this.filter = new ContractFilter ();
    }

    public List<Contract> inactivatedBy (
                        List<Contract> signedContracts ) {

        List<Contract> originalContracts = findOriginalContracts (
                                                signedContracts );

        if (originalContracts.isEmpty()) {
```

```
            return new List<Contract> ();
        }

        Map<Id, Contract> indexedOriginalContracts =
                        this.filter.indexById (originalContracts);

        for (Contract signedContract : signedContracts ) {

            Contract originalContract =
                    indexedOriginalContracts.get (
                         signedContract.OriginalContract__c);

            originalContract.Status = 'Inativado';

        }

        return repository.save ( originalContracts );

    }

    private List<Contract> findOriginalContracts (
                          List<Contract> signedContracts ) {

        List<String> originalContractIds =
                        filter.extractOriginalContractIds (
                                              amendmentContracts );

        return repository.findByIds ( originalContractIds );

    }

}
```

Como vamos testar a classe `InactivateOriginalContractEnricher` ? Antes de responder, vamos ver quais pontos devem ser observados para realizar os testes "mockados" de uma forma mais genérica.

Sempre comece analisando as dependências, se elas estão disponíveis para que possam ser *setadas* pela classe de testes. Em

seguida, avalie se essas dependências estão prontas para a sobrescrita. Com esses pontos em mente, recomendo que inicie pelas dependências que fazem operações com o banco de dados, tanto *SOQL* quanto *DML* — no nosso caso, são as classes do tipo `Repository`.

Após estabelecer essa estrutura de pensamento, volte para a classe apresentada anteriormente e responda à pergunta de como vai testá-la. Não prossiga até tentar responder à pergunta :)!

Pronto! Acredito que você chegou à resposta de que devemos "mockar" a classe `ContractRepository`, simulando o comportamento de `findByIds` e `save`.

> **#se-liga-na-info** 😎
>
> Perceba que a resposta para a pergunta ficou mais genérica, utilizando "simulando o comportamento". Isso porque agora teremos duas estratégias de realização do teste "mockado": uma através de classes ***Fake/Stub***, que são as classes `XXXMock`, e outra através do framework *Mocker*.

Mockando a Enricher com Fake/Stub

Vamos simular os comportamentos de nossa `ContractRepository` através da `ContractRepositoryMock`:

```
@isTest
public class ContractRepositoryMock
                        extends ContractRepository {

    String payload;

    public ContractRepositoryMock (String payload) {
        this.payload = payload;
```

```
    }

    override
    public  List<Contract> findByIds ( List<String> ids ) {
        return (List<Contract>)
                    JSON.deserialize (this.payload
    }

    override
    public  SObject save ( SObject record ) {

        if (record.Id == null) {
            IdGenerator.generate(record);
        }

        return record;
    }

}
```

O método sobrescrito `save` agora ficou um pouco mais esperto que os exemplos apresentados anteriormente. Isso porque o comportamento esperado é que ele possa criar e atualizar um registro. Então o método sobrescrito só vai gerar um `Id` caso não seja informado.

Com o nosso *Mock* pronto, podemos implementar a nossa classe de testes que, através do método `givenAmendmentContractsWhenSignedThenInactivateOriginalContracts`, avaliará o cenário de sucesso de inativação do contrato:

```
// given

String signedContractsPayload =
      ' [{ "Id": "8008c000001dSGNAA2",'
      + '  "Status": "Assinado",'
      + '  "AccountId": "0018c000028ktwCAAQ",'
      + '  "OriginalContract__c": "8008c000001dSGIAA2"'
      + '}]';
```

```
List<Contract> signedContracts =
        ContractFixtureFactory.fromJsonArray (
                              signedContractsPayload);

String originalContractsPayload =
                '[ { "Id": "8008c000001dSGIAA2",'
             + '    "Status": "Assinado",'
             + '    "AccountId": "0018c000028ktwCAAQ"'
             + '  }]';

ContractRepository mockedRepository =
        new ContractRepositoryMock (originalContractsPayload);

InactivateOriginalContractEnricher enricher
            = new InactivateOriginalContractEnricher ();

enricher.setRepository ( mockedRepository );


Test.startTest();

// when
List<Contract> inactivatedContracts;

inactivatedContracts = enricher.inactivatedBy (
                                     signedContracts );

Test.stopTest();

Assert.isFalse ( inactivatedContracts.isEmpty() );

Assert.areEqual ('Inativo', inactivatedContracts.get(0).Status );
```

Note que, em nosso `assert`, não temos a necessidade de avaliar o `Id`, pois se trata de um fluxo de atualização e, com isso, só precisamos avaliar se os contratos retornados estão como inativos.

Em vários outros cenários, poderíamos utilizar a mesma estratégia de ***Mock***, injetando uma nova instância da classe base, redeterminando o seu comportamento.

Mockando a Enricher com o framework *Mocker*

Vimos que, para testar de forma "mockada", temos a necessidade de criar classes `XXXMock`, que, seguindo o padrão apresentado, são bem simples, mas ainda são classes que precisamos criar. Através do framework *Mocker*, não precisamos criar as classes de *Mock*, pois o *Mock* será realizado de forma dinâmica.

Veja o mesmo cenário de teste implementado com o *Mocker*:

```
// --- Arrange
// given
Mocker mocker = Mocker.startStubbing();

ContractRepository mockedRepository = (ContractRepository)
                        mocker.mock(ContractRepository.class);

String originalContractsPayload = '[{ "Id": "8008c000001dSGIAA2",
"Status": "Assinado", "AccountId": "0018c000028ktwCAAQ" }]';


List<Contract> originalContracts =
                    ContractFixtureFactory.fromJsonArray (
                                    originalContractsPayload );

// gravando o comportamento do mock
mocker.when( mockedRepository.findByIds( new List<String>{''}) )
                        .withAnyValues()
                        .thenReturn(originalContracts)
                        .shouldBeCalledOnce();


String expectedContractsPayload = '[{ "Id": "8008c000001dSGIAA2",
"Status": "Inative", "AccountId": "0018c000028ktwCAAQ" }]';
```

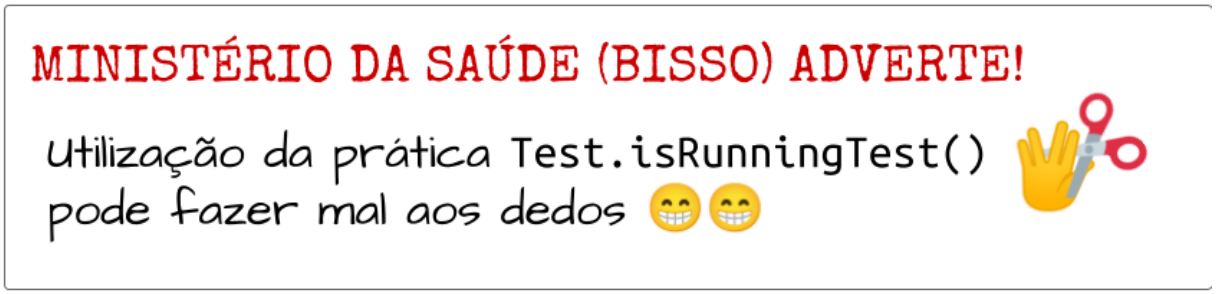
```
List<Contract> expectedContracts =
                ContractFixtureFactory.fromJsonArray (
                                    expectedContractsPayload );

mocker.when( mockedRepository.save( expectedContracts ))
                                .thenReturn(expectedContracts)
                                .shouldBeCalledOnce();

mocker.stopStubbing();

InactivateOriginalContractEnricher enricher
            = new InactivateOriginalContractEnricher ();

enricher.setRepository ( mockedRepository );

// --- Act
// When
List<Contract> inactivatedContracts;

inactivatedContracts = enricher.inactivatedBy (
                                    amendmentContracts );

// --- Assert
// Then
mocker.assert();

Assert.isFalse ( inactivatedContracts.isEmpty() );

Assert.areEqual ('Inativo', inactivatedContracts.get(0).Status);
```

Esse cenário ficou mais simples e rápido de produzir. Fiz questão de trazer as duas alternativas de *Mock* para que você possa compará-las mais de perto. A questão aqui não é se uma é melhor do que a outra, mas, sim, qual a alternativa que posso usar para testar de forma mais rápida e eficiente o código produzido.

17.2 Utilize Mock em vez de Test.isRunningTest()

Em algumas ocasiões nos deparamos com um trecho de código que não pode ser executado em escopo de teste, e é muito comum que desenvolvedores utilizem o método `Test.isRunningTest()` para avaliar se estamos em escopo de teste ou não. Essa é uma **má prática** de desenvolvimento, dado que está misturando código de teste com código produtivo.

Uma alternativa a esse problema/prática é a utilização de *Mocks* para substituir o comportamento durante a execução do teste.



Para exemplificar esse problema, vou trazer um exemplo real de aplicabilidade de um cenário onde é comum ser aplicado o método `Test.isRunningTest()` e como o alteramos para utilizar *Mocks*.

Classe assíncronas que utilizam `Queueable` podem realizar encadeamento de chamadas, ou seja, uma `Queueable` pode chamar outra `Queueable`, porém não podemos executar esse encadeamento durante os testes, gerando o erro `System.AsyncException: A profundidade máxima da pilha foi atingida.`

Agora veja o exemplo do código a seguir, onde, no método `execute`, avaliamos se a execução está em escopo de teste, `Test.isRunningTest()`, e interrompemos o processamento. Esse cenário será apresentado pela classe `ChainQueueable`, que é encadeada até que não tenha mais logs para processar:

```
public class ChainQueueable implements Queueable {

    private static final Integer DEFAULT_CAPACITY = 10;

    List<Log__c> logs;

    AsyncLoggerRepository repository;

    private ChainQueueable() {
        repository = new AsyncLoggerRepository();
    }

    public ChainQueueable(List<Log__c> logs) {
        this();
        this.logs = logs;
    }

    public void execute (QueueableContext context) {

        repository.save ( getFirstChunk() );

        List<Log__c> newEnqueueLogs = splitToNextChain();

        if (newEnqueueLogs.isEmpty()) return;

        // aqui o código de teste mistura com o produtivo
        if (Test.isRunningTest()) return;

        System.enqueueJob( new ChainQueueable (newEnqueueLogs));

    }

    private List<Log__c> splitToNextChain () {

        if (logs.size() <= DEFAULT_CAPACITY) {
            return new List<Log__c>();
        }

        List<Log__c> newChainLogs = new List<Log__c>();
```

```
        Integer i = DEFAULT_CAPACITY;

        for (  ; i < logs.size(); i++) {
            newChainLogs.add ( logs.get(i));
        }

        return newChainLogs;
    }

    private List<Log__c> getFirstChunk() {

        if (logs.size() <= DEFAULT_CAPACITY) return logs;

        return Lists.split(logs, DEFAULT_CAPACITY).get(0);

    }
}
```

Note que o método `execute` salva o lote de logs e, em seguida, realiza o encadeamento da próxima camada, caso não esteja em um escopo de testes. Para driblar esse ponto, podemos encapsular o código que realiza o encadeamento propriamente dito por meio do método `enqueueNextJob` , que basicamente realiza a chamada do comando `System.enqueueJob( new ChainQueueable (newEnqueueLogs));` :

```
public virtual class ChainQueueable implements Queueable {

    // trecho de código omitido

    public void execute (QueueableContext context) {

        repository.save ( getFirstChunk() );

        List<Log__c> newEnqueueLogs = splitToNextChain();

        if (newEnqueueLogs.isEmpty()) return;

        enqueueNextJob( newEnqueueLogs );
```

```
    }

    virtual
    protected void enqueueNextJob (
                            List<Log__c> newEnqueueLogs ) {

        System.enqueueJob( new ChainQueueable (newEnqueueLogs));

    }

}
```

Para testar essa classe, não poderemos utilizar o framework *Mocker*. Isso porque não conseguimos "mockar" parcialmente com ele, não sendo possível substituir somente a execução do `enqueueNextJob`. Então vamos criar a classe `ChainQueueableMock`, que deverá sobrescrever o método `enqueueNextJob`, driblando o comportamento indesejado:

```
public class ChainQueueableMock extends ChainQueueable {

    public ChainQueueableMock (List<Log__c> logs) {
        super(logs);
    }

    override
    public void enqueueNextJob (List<Log__c> newEnqueueLogs ) {
        System.debug ('enqueue next job');
    }

}
```

O método que foi sobrescrito está somente logando um texto no console, e esse log não tem um propósito de verificação. Ele é somente uma aferição visual para você, e poderia até ser removido.

Veja a seguir nossa classe de teste onde executamos a classe `ChainQueueableMock`:

```
static List<Log__c> buildLogs (Integer size) {

    List<Log__c> logs = new  List<Log__c>();

    for (Integer i = 0; i < size; i++) {
        logs.add ( new Log__c( Message__c = 'Message ' + i
                              , Payload__c =  'Payload ' + i ) );
    }

    return logs;
}

@isTest
static void givenSomeLogsWhenEnqueueThenStoreLogs() {

    List<Log__c> logs = buildLogs(50);

    Test.startTest();

    System.enqueueJob(new ChainQueueableMock(logs));

    Test.stopTest();

    Integer count  = Database.countQuery(
                        'SELECT COUNT() FROM Log__c');

    System.assertEquals(10, count);

}
```

Ao executar esse método de teste, obtemos uma excelente cobertura de 93%. Veja o *report* obtido com essa execução:

```
=== Apex Code Coverage by Class
CLASSES                 PERCENT  UNCOVERED LINES
______________________  _______  _______________

AsyncLoggerRepository   38%      4,5,6,14,15
ChainQueueable           93%      67,69
```

Como temos diversas técnicas em mãos, podemos escolher a que melhor se encaixa em nosso cenário. Nesse caso, foi a utilização de uma classe *fake* para simular o comportamento.

Outra característica desse teste é que ele é **híbrido**, onde simulamos somente o encadeamento de chamadas e executamos o restante de forma integrada. Cenários híbridos devem ser observados com bastante atenção para que não escape um dado *fake* que funcione em sua **Org** e, em outra, acabe dando erro de `CROSS REFERENCE ID`.

Mas você deve estar pensando se seria possível testá-la totalmente "mockada", e a resposta é **sim**. Veja o código a seguir onde trazemos uma alternativa utilizando o framework, mas desta vez não executaremos a classe `ChainQueueable` original, e sim a nossa versão ***fake***:

```
@isTest
static void givenSomeLogsWhenMockEnqueueThenStoreFakeLogs() {

    Mocker mocker = Mocker.startStubbing();

    List<Log__c> logs = buildLogs(50);

    List<Log__c> expectedLogs = (List<Log__c>)
                                    Lists.split(logs, 10)
                                                        .get(0)

    AsyncLoggerRepository mockedRepository;

    mockedRepository = (AsyncLoggerRepository)
                            mocker.mock (
                                    AsyncLoggerRepository.class);

    mocker.when( repository.save( new List<Log__c>()) )
            .withAnyValues()
            .thenReturn( expectedLogs )
            .shouldBeCalledOnce();
```

```
    mocker.stopStubbing();

    ChainQueueable chainQueueable = new ChainQueueableMock (logs);

    //método adicionado para injeção de dependência
    chainQueueable.setRepository(mockedRepository);

    chainQueueable.execute(null);

    mocker.assert();

}
```

Versatilidade é a palavra-chave! Agora temos ampla capacidade de escolher a melhor alternativa de teste. No código anterior, utilizamos o framework juntamente com a nossa classe ***fake***, onde não executamos o encadeamento e simulamos a persistência dos dados. "Lindo D+"!

17.3 Mockando classes com escopo estático

A plataforma Salesforce tem algumas características para APIs que ficam mais próximas do front-end. Isso acontece com classes `@RestResource` e os métodos anotados `@AuraEnabled` e `@InvocableMethod`. Essas implementações utilizam métodos estáticos como estrutura de sua implementação. Então você pode utilizar a mesma estratégia de testes aplicada para o `@RestResource` para os outros tipos.

Métodos estáticos não suportam herança e, com isso, não suportam polimorfismo — e existe uma tendência natural de que você fique tentado a implementar outros métodos também estáticos. Vamos então rever a implementação da classe de `@RestResource`, que já traz um "pulo do gato", onde todas as dependências são declaradas como variáveis também estáticas:

```
@RestResource (urlMapping='/api/orders/*')
global class OrderController {

    public static OrderInboundService orderInboundService;

    static {
        orderInboundService = new OrderInboundService();
    }

    @HttpPost
    global static void create (
                PurchaseOrderInboundRequest purchaseOrder ) {

        RestResponseBuilder responseBuilder =
                new RestResponseBuilder(RestContext.response);

        PurchaseOrderInboundResponse purchaseOrderResponse;

        try {

            purchaseOrderResponse =
                 orderInboundService.save(purchaseOrder);

            responseBuilder.withBody(purchaseOrderResponse);

        } catch (DmlException e) {

            responseBuilder.withError(e);

        } catch (Exception e) {

            responseBuilder.withError(e);

        }

    }

}
```

O destaque dessa implementação está expresso neste pequeno bloco de código, que nos possibilita alterar a implementação da classe `OrderInboundService`:

```
    public static OrderInboundService orderInboundService;

    static {
        orderInboundService = new OrderInboundService();
    }
```

Note que essa implementação utiliza uma variável estática `OrderInboundService orderInboundService`, que é inicializada em um bloco também estático. Essa estratégia possibilita *injetar* uma instância de `OrderInboundService` em nossas classes de teste. Caso essa dependência fosse criada dentro do método estático, não poderíamos passar uma nova instância e, com isso, não seria possível trabalhar com testes "mockados".

Temos então um padrão de construção de classes, que utilizam métodos estáticos em sua implementação. Lembrando que podemos aplicar em classes `@RestResource`, em métodos anotados `@AuraEnabled`, `@InvocableMethod`, entre outras.

#se-liga-na-info 😎

Sempre prepare suas classes para que possam ser mais bem testadas, com a capacidade de injetar as suas dependências. Com isso, podemos aplicar diversas técnicas de teste.

Vamos implementar o nosso teste injetando as instâncias "mockadas" de classes do tipo **Repository**. Com isso, a classe `OrderInboundService` vai trabalhar com tudo "mockado".

```
@isTest
static void givenPurchaseOrderWhenCreateThenReturnStoredOrder() {
```

```
// given PurchaseOrder
RestContext.response = new RestResponse();
RestContext.request = new RestRequest();

PurchaseOrderInboundRequest purchaseOrder =
                        PurchaseOrderFixtureFactory
                                    .newPurchaseOrder();

//jump of the cat ;)
OrderController
    .orderInboundService.setOrderRepository(
                new OrderRepositoryDmlExceptionMock());

OrderController
    .orderInboundService.setOrderLineItemRepository(
        new OrderLineItemRepositoryMock());

OrderController
    .orderInboundService.setOrderSubItemRepository(
        new OrderSubItemRepositoryMock());

//when
Test.startTest();

OrderController.create ( purchaseOrder );

Test.stopTest();

// then return stored lead
System.assertEquals( 200
            ,  RestContext.response.statusCode);

PurchaseOrderInboundResponse purchaseOrderResponse;

purchaseOrderResponse =
    PurchaseOrderFixtureFactory.fromJson(
        RestContext.response.responseBody.toString());

System.assert(purchaseOrderResponse.id != null);
```

```
}
```

Essa implementação se assemelha ao código de *Mock* da própria classe `OrderInboundService`, onde injetamos os `Mocks` e depois fazemos os respectivos `asserts`. A diferença é que utilizamos a variável estática para injetá-los.

Outra estratégia seria "mockar" completamente a classe `OrderInboundService`, dado que ela já deveria estar completamente testada. Com isso, poderíamos considerar somente seu método `save` e os cenários que envolvem sucesso e falha. Nesta nova implementação, vamos utilizar o *Mocker* como estratégia de teste:

```
@isTest
static void givenPurchaseOrderWhenCreateThenReturnStoredOrder() {

    // given PurchaseOrder
    RestContext.response = new RestResponse();
    RestContext.request = new RestRequest();

    PurchaseOrderInboundRequest purchaseOrder =
                            PurchaseOrderFixtureFactory
                                        .newPurchaseOrder();

    PurchaseOrderInboundResponse response =
                        PurchaseOrderFixtureFactory
                                .newPurchaseOrderResponse();

    Mocker mocker = Mocker.startStubbing();

    OrderInboundService mockedService;

    mockedService = (OrderInboundService)
                        mocker.mock (OrderInboundService.class);

    mocker.when( mockedService.save (purchaseOrder) )
        .thenReturn ( response )
        .shouldBeCalledOnce();
```

```
    mocker.stopStubbing();

    //jump of the cat ;)
    OrderController.orderInboundService = mockedService;

    //when
    OrderController.create ( purchaseOrder );

    // then return stored lead
    Assert.areEqual( 200
                ,  RestContext.response.statusCode);

    PurchaseOrderInboundResponse purchaseOrderResponse;

    purchaseOrderResponse =
        PurchaseOrderFixtureFactory.fromJson(
            RestContext.response.responseBody.toString());

    Assert.isTrue(purchaseOrderResponse.id != null);

}
```

A implementação anterior trata somente do fluxo de sucesso, ficando ligeiramente menor. Devemos ainda criar os cenários de falha para os fluxos de `Exception` e `DmlException`. Observe no código a seguir como podemos simular, de forma simples, o fluxo de erro não esperado:

```
// exception utilizada para simular uma exception genérica
public class UnHandledException extends Exception  {
}

@isTest
static void givenInvalidPurchaseOrderWhenProcessThenReturnHttpError() {

    // given PurchaseOrder
    RestContext.response = new RestResponse();
    RestContext.request = new RestRequest();
```

```
PurchaseOrderInboundRequest purchaseOrder =
                        PurchaseOrderFixtureFactory
                                    .newPurchaseOrder();

Mocker mocker = Mocker.startStubbing();

OrderInboundService mockedService;

mockedService = (OrderInboundService)
                    mocker.mock (OrderInboundService.class);

mocker.when( mockedService.save (purchaseOrder) )
    .thenThrow (
        new UnHandledException ('Erro não esperado')
    );

mocker.stopStubbing();

//jump of the cat ;)
OrderController.orderInboundService = mockedService;

//when
Test.startTest();

OrderController.create ( purchaseOrder );

Test.stopTest();

// then return stored lead
System.assertEquals( 400
            ,  RestContext.response.statusCode);

RestResponseError responseError;

responseError =
    RestResponseError.fromJson(
        RestContext.response.responseBody.toString());
```

```
    System.assert(responseError != null);
    System.assertEquals( 'Erro não esperado'
                    ,  responseError.error);

}
```

A implementação anterior demonstra como podemos direcionar os nossos cenários de teste para conseguir cobrir de maneira adequada todos os fluxos possíveis. Para o fluxo de `DmlException`, basta trocar a implementação do método `thenThrow()` passando uma instância de `DmlException`.

O ponto-chave dessa estratégia de teste é que as classes que necessitam implementar métodos estáticos determinem suas dependências como variáveis também estáticas, como o **apresentado novamente** a seguir:

#se-liga-na-info 😎

"Vale a pena ver de novo"

```
public static OrderInboundService orderInboundService;

static {
    orderInboundService = new OrderInboundService();
}
```

17.4 Mockar ou testar de forma integrada?

Este capítulo trouxe diversos cenários para testes "mockados", lembrando que sempre devemos preparar nossas classes para que possam ser "mockadas". Evite a utilização de métodos estáticos, pois não será possível redeterminar o comportamento. Faça com que

seus *FixtureFactories* sempre tenham um método para desserializar um *Payload* JSON. Isso facilita bastante na utilização das classes de teste.

Reconheço que, no começo, você terá bastante dificuldade para iniciar essa nova forma de testar, mas não desista, só faça! E terá também uma dúvida — constante — se você deve testar de forma integrada ou "mockada". Na verdade não existe um certo e errado, temos que utilizar o melhor dos dois cenários. O único problema do teste integrado é o tempo de execução, por isso deve ele ser usado com atenção.

De tudo que vimos até agora, vou tentar apoiá-lo ou apoiá-la trazendo opções de teste que você pode utilizar, mas você deve buscar o contexto do seu projeto e da Org como um todo. Então tenha uma visão completa, não somente do seu projeto em específico.

Classe de Repository

Faça testes integrados, pois o único propósito de classes do tipo `Repository` é acessar a camada mais baixa do sistema. Eles ajudarão a identificar também vários problemas em regras de validação, fluxos, *process builders*, entre outros. Utilize as *FixtureFactories* para criar dados de teste reutilizáveis.

Triggers e TriggerHandlers

Classes desse tipo têm o único propósito de ouvir eventos de atualização do banco de dados. Nesse cenário, recomendo que 100% dos testes sejam integrados. Assim como as classes do tipo `Repository`, o único propósito é a interação com o banco.

Filters

100% "mockadas", são classes que não necessitam de dados persistidos, pois recebem listas e retornam outras listas ou um

agrupamento ou indexação de dados.

Enrichers

Classes de testes de `Enricher` devem ser 100% "mockadas". Caso tenha necessidade de testar de forma integrada, veja se o teste realizado pela `Trigger` não cobre essa necessidade.

Transformers

As `Transformers` são usadas no escopo de integração sistêmica, com o propósito de converter os dados transitados entre os sistemas. Com isso, devemos realizar testes 100% "mockados" e com **asserção em todas as linhas de transformação**. Lembre-se de que é onde ocorrem os problemas!

OutboundServices

Classes de `OutboundServices` são aquelas que criamos para realizar uma interação com um sistema externo. Seus testes necessitam de `Test.setMock` por natureza, mas não confunda o `Test.setMock` com nossos **dublês de teste (*Mocks*)**. Há similaridade, mas são estratégias diferentes.

Recomendo, inicialmente, que seus testes sejam 100% "mockados". Mas se sua integração realizar operações de *DML* e nesse fluxo houver *flows*, *process builders* ou *workflows*, sugiro que faça pelo menos um fluxo integrado.

InboundServices

Quando estamos expondo uma API customizada, utilizamos as classes `@RestResources`, os nossos `InboundServices`. Elas dever ser basicamente "mockadas", mas, assim como mencionado nas `OutboundServices`, podemos ter cenários de teste integrado com a

intenção de cobrir os fluxos parametrizados da solução, aumentando a robustez do teste.

Future, Queues, Schedules

Podemos trabalhar basicamente "mockados" com esses cenários.

Batches

Classes `Batches` podem ser "mockadas", pois podemos testar o comportamento do método `execute` simulando todos os fluxos de forma isolada, sem a necessidade de montar um `setup` monstruoso para a execução de todos os cenários. A recomendação de testes integrados permanece a mesma: avalie o contexto; se necessário, faça ao menos um teste integrado.

17.5 Considerações finais

Em todos os cenários de testes "mockados", devemos preparar nossas classes para que possamos injetar a nova dependência "mockada". Com as técnicas apresentadas neste capítulo, você terá capacidade de "mockar" todas as implementações apresentadas.

Note que, em quase todos os cenários apresentados, basicamente não descartamos a hipótese de realizar **testes integrados**, mas sempre a minimizamos. Dessa forma, conseguimos equilibrar o tempo de execução, que é um dos principais princípios de testes.

Esses princípios devem nortear você para sempre buscar uma base de código limpa, com capacidade de escalar funcionalmente — e para isso deve ser cuidada e evoluída constantemente.

Acredito que, com todas essas técnicas aplicadas, você terá a capacidade de evoluir a sua base de código e deixá-la 100% testada — esse é o sonho de todo cliente Salesforce :).

Capítulo 18
Conclusão

Desenvolver de forma profissional não é uma tarefa fácil: requer muita atenção, dedicação e prática constante. Não será de primeira, você ainda errará bastante. Utilize este livro como uma referência para seu desenvolvimento e, a cada implementação, volte e revise o conteúdo.

A palavra aqui é **prática**: fazer buscando a melhoria contínua. Sugiro que utilize os desafios do TrailHead como exemplo de cenário, mas não de implementação, dado que o foco da plataforma não é mostrar a você a melhor forma de desenvolver.

Prepare-se, tenha pleno conhecimento de padrões de desenvolvimento, busque outras referências. Tudo o que está descrito aqui não foi inteiramente criado por mim — na verdade, quase nada. Este livro é um condensado de práticas de desenvolvimento de software, não somente Salesforce. Com ele, você aplicou padrões de ***clean code***, princípios do S.O.L.I.D, padrões do GOF, padrões de testes, entre outros.

Não pare neste livro, faça você também uma curadoria! Zele, cuide, dê atenção ao seu conhecimento. Reserve um tempo para ler, praticar e aprender. E compartilhar! Você aprende mais quando o conhecimento é compartilhado!

18.1 Indo além

Seguem alguns dos livros que mudaram a minha maneira de desenvolver software — e acredito que também mudarão a sua:

- *Padrões de Projetos: Soluções Reutilizáveis de Software Orientados a Objetos*, de Erich Gamma, Richard Helm, Ralph Johnson e John Vlisside (GOF Design Patterns);
- *Código Limpo*, do Robert Martin (o *Uncle Bob*);
- *Trabalho Eficaz com Código Legado*, do Michael Feathers;
- *Growing Object-Oriented Software, Guided by Tests*, de Steve Freeman e Nat Pryce;
- *Refatoração: Aperfeiçoando o Design de Códigos Existentes*, por Martin Fowler;
- *Domain-Driven Design: Atacando as Complexidades no Coração do Software* (DDD), do Eric Evans;
- *Implementando Domain-Driven Design*, do Vaughn Vernon.

Não se limite a somente conhecimento técnico! Somente isso não fará você crescer. Trabalhe áreas de conhecimento ligadas ao comportamental, conhecidas como *soft skills*. Segue uma listinha de livros que tenho certeza de que vão exponencializar seu conhecimento:

- *Os 7 Hábitos das Pessoas Altamente Eficazes*, do Stephen R. Covey;
- *Mindset*, da Carol S. Dweck;
- *Essencialismo*, do Greg McKeown;
- *Micro-hábitos*, do B. J. Fogg;
- *Os segredos da mente milionária*, do T. Harv Eker;
- *Fora de série — Outliers*, do Malcolm Gladwell.

Obrigado!

Quero agradecer a você por chegar até aqui. Espero ter agregado um pouco de conhecimento e ajudado a melhorar a forma como você desenvolve software. Não deixe de me seguir em:

- LinkedIn: https://www.linkedin.com/in/ercarval-bisso
- Instagram: https://www.instagram.com/eduardo.bisso/

Para fechar, só *code*! Só faça!

Todos os códigos deste livro estão separados por *branch* no repositório do GitHub (https://github.com/ercarval/salesforce-book-snippets) com uma formatação elegante.

Forte abraço e até logo!

Capítulo 19
Referências

BECK, Kent. *TDD — Desenvolvimento Guiado por Testes*. 1. ed. Porto Alegre: Bookman, 2010.

DALIO, Ray. *Principles*: Life and Work. New York: Simon & Schuster, 2017.

EVANS, Eric. *Domain-Driven Design*: Atacando as Complexidades no Coração do Software. 3. ed. Rio de Janeiro: Alta Books, 2016.

FEATHERS, Michael C. *Trabalho Eficaz com Código Legado*. 3. ed. Porto Alegre: Bookman, 2013.

FOWLER, Martin. *Padrões de Arquitetura de Aplicações Corporativas*. 1. ed. Porto Alegre: Bookman, 2006.

FREEMAN, Steve; PRYCE, Nat. *Growing Object-Oriented Software, Guided by Tests*. 1. ed. Boston, MA: Addison-Wesley Longman, 2009.

GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. *Padrões de Projetos*: Soluções reutilizáveis de software orientados a objetos. Porto

Alegre: Bookman, 2007.

MARTIN, Robert C. *Código Limpo*: Habilidades práticas do Agile Software. 1. ed. Rio de Janeiro: Alta Books, 2009.

MESZAROS, Gerard. *XUnit Test Patterns*: Refactoring Test Code. Boston, MA: Addison-Wesley, 2007.